Ref. Ares(2017)3949215 - 08/08/2017

European Commission

**TagItSmart**
**Smart tags driven service platform for enabling ecosystems of connected objects**

**Grant agreement 688061**

# Service composition platform description (CaaS)

| | |
|---|---|
| Deliverable ID: | **D4.2** |
| Deliverable Title: | **Service composition platform description (CaaS)** |
| Revision #: | **1.0** |
| Dissemination Level: | **Public** |
| Responsible beneficiary: | **UNIS** |
| Contributing beneficiaries: | **UNIS, UNIPD, UPC, SIE, EVT, DNET** |
| Contractual date of delivery: | **30.06.2017** |
| Actual submission date: | **08.08.2017** |

Start Date of the Project: 1 January 2016 Duration: 36 Months

# Contents

***Section 1 - Introduction ..... 3***

***Section 2 - Related work ..... 4***

***Section 3 - CaaS framework overview ..... 6***

**3.1 CaaS framework processes ..... 6**

3.1.1 Template Creation ..... 7

3.1.2 Template Resolution ..... 8

3.1.3 Template Execution ..... 8

**3.2 CaaS framework example ..... 8**

3.2.1 Template Creation ..... 9

3.2.2 Template Resolution ..... 11

3.2.3 Template Execution ..... 12

***Section 4 - CaaS components ..... 13***

**4.1 Repositories for exposing information for service composition ..... 13**

4.1.1 Service Template Repository ..... 13

4.1.2 Service Repository ..... 23

4.1.3 VE Repository ..... 28

4.1.4 User Repository ..... 30

4.1.5 SmartTag Repository ..... 31

**4.2 Service setup and runtime execution components ..... 32**

4.2.1 Service Manager ..... 32

4.2.2 Workflow enforcer ..... 37

***Section 5 - Service Composition in Action ..... 40***

***Section 6 - Conclusions ..... 50***

***Section 7 - References ..... 51***

## Section 1 - Introduction

CaaS in TagItSmart focuses on enabling developers to define reusable workflows based on semantically defined components. The role of the service composition layer/part in the TagItSmart platform architecture, following that process, is to subsequently allow users of the platform to issue service requests that are then processed by the platform, resolved in terms of sub-services that need to be "glued together" to support the overall service request and are then executed.

In order for this process to take place, a series of repositories are required to first of all expose the services that are available in the platform so they can be discovered, configured and triggered appropriately so that they can be re-used. In addition, components that intercept the service requests and are able to break them down to services that are needed are also required. In this process, templates that guide this process of identifying what a service request needs in terms of sub-services are also needed as well user information. The latter both for making sure that only authorised users are allowed to access the platform and issue service requests as well as be served through the platform (e.g. an FC-scanner user to be able to use a decoding or stream processing component that is hosted in the TagItSmart platform) but also for identifying appropriate resources based on ownership, location and other availability criteria that need to be involved during a service request execution.

As an example, if a service request involves displaying as an outcome of it a relevant notification to an end-user (owner of an FC-scanner) based on scanning a SmartTag, then the appropriate devices owned by this end-user and which have message display capabilities will need to be identified and involved during the service execution. Finally a component that executes during runtime all the steps needed in a fulfilling a service request is needed; in TagItSmart this component is termed Wokflow Enforcer.

In the rest of this document, following a state of the art analysis, we present in detail all the involved components and repositories that are involved in the above mentioned steps giving also, where appropriate, specific examples of operations and entries in the relevant databases. In more detail, in Section 2 we review recent and ongoing EU initiatives that have also worked on developing service composition platforms. Then in Section 3 we give an overview of the TagItSmart composition as a service (CaaS) framework, presenting the architectural components that are involved in this framework and the processes they need to support. In Section 4 we present in details these components and databases while in Section 5 we give an "end-to-end" example illustrating the whole lifecycle of a service request, from how it is inserted as something the platform supports by a developer, to how it is invoked and all the steps involved until its runtime execution.

Finally in Section 6 we conclude this deliverable giving directions for the next steps in the corresponding task (T4.2).

## Section 2 - Related work

There has been a number of EU projects that were recently active or are still active, which have worked on the idea of service composition.

IoT-A [1] was a pioneer in the IoT area setting the foundations upon which many follow-up projects were based. IoT-A was one of the very first projects that acknowledged the merit of having a service composition layer in a platform architecture to allow the re-use of underlying platform capabilities for supporting diverse service requests. To support this, a Service Organisation Functional Block was introduced to support the operations of service composition. IoT-A however stayed at a conceptual level, not providing concrete implementations but rather staying at a high level with the purpose of specific “IoT-A compliant” implementations to be provided by other follow-up projects.

iCore [2] was one of the first follow-ups to IoT-A that attempted to map generic specifications to specific implementations of needed components. For that purpose, iCore adopted a 3-layer architecture (Virtual Object level, Composite Virtual Object level and Service level) where the first two layers were exposing capabilities and services while the Service level was in charge of finding the most suitable combinations for them. One of the drawbacks of the iCore project’s approach was that many service layer implementations were developed in silos meaning that even within the boundaries of the project, interoperability was lacking with different interface specifications at a per-use case level.

iKaaS [3] developed a cloud platform where, contrary to iCore, also underlying cloud platfrom capabilities were taken into account for service composition (e.g. to ensure that using a service does not overload the physical host where the service runs). To a large extent iKaaS followed the same principles as iCore but didn’t repeat the same mistake of creating scattered and not inter-operable service level implementations. However, and similar to iCore, one shortcoming of iKaaS has been the lack of a production grade service execution engine (see next section and section 4.2.2). That meant that for service orchestration during the service execution, the corresponding task of “keeping an eye” on how well the service execution progresses was delegated to service components themselves meaning that individual components (e.g. data analytics) were also tasked with implementing additional functionalities not related to their core task.

IoT.est [4] was a follow-up of IoT-A, executed on the same time with iCore. The project had a targeted full life-cycle view of semantically annotated IoT services, working on both design time and run-time of services. The project identified atomic and composed services, providing meanings to determine conditions under which services are selected, deployed and executed in a knowledge driven manner. Special focus was given to a test generation used for performance monitoring and eventual adaptation or generation of new service workflows.

COSMOS [5] was an EU project in the Smart Cities context. Main foci have been set along device to cloud security solutions and on the contextual ability of things to learn from the environment where they are deployed. This capacity targeted also socialy-like behavior networking of things apart from individual self-optimization. The project achieved relevant demonstrations on two large use cases for public transportation and districtual heating. COSMOS also provided an extended semanting profile of things at service level but missing the description needed on resource level.

The objective of the COMPOSE project [6] was to create an IoT Platform that could easily expose connected things as services in the cloud and create interactions among them. It implemented a fully fledged IoT platform, considering aspects from the virtualization of objects to the actual deployment in the cloud of user services. COMPOSE platform relied on a cloud deployment of a workflow engine (based on Node-Red [7]) but workflows were specific for each case, limiting reusability and dynamic allocation based on semantic descriptions. COMPOSE produced a W3C [8] recommendation on a model to describe things on the Web, the Web Thing model, which contributed to kickstart current work towards standardisation of the Web of Things, which is as well being used and extended in TagItSmart.

In conclusion, while these projects set the foundations for service composition in the IoT ecosystem, they had limitations in terms of reusability due to lack of proper semantic representations (i.e. “hard coded” service requests) as well as scattered and use case specific implementations. This is in sharp contrast to TagItSmart where apart from following an approach with semantic support to foster re-usability, also a unique project wide implementation of the service layer is underway.

In addition, this service layer tightly integrates to a production level service execution engine (workflow enforcer) provided by both the EVT engine as well as through the Microsoft Azure platform. In that sense, TagItSmart also tries to re-use and integrate well-established components rather than developing a “just one yet” execution engine with limited capabilities, as has been the case in some of the previous inititiatives.

## Section 3 - CaaS framework overview

In deliverable D1.2 [9] we described the functional reference architecture of TagItSmart and the main information flows that will allow us to explain how the CaaS framework is designed. From the functional architecture, the following figure highlights the components that are the building blocks of the CaaS framework.

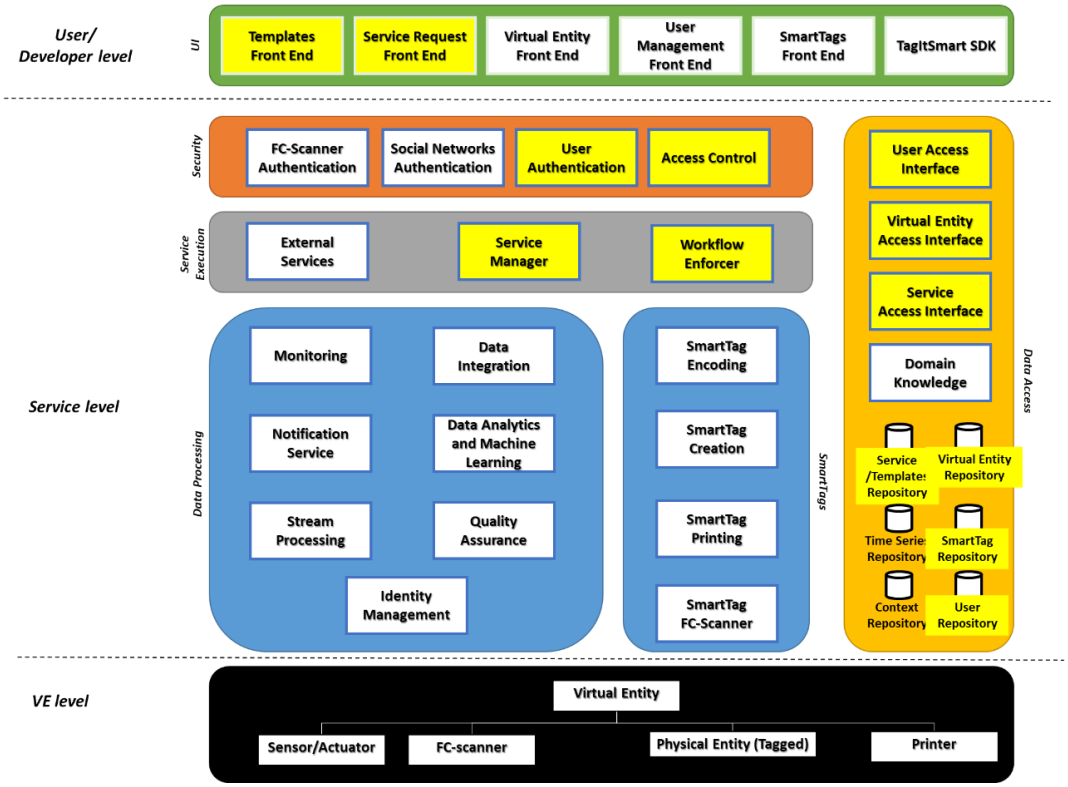


**Figure 1. Components of the TagItSmart platform involved in Service Composition**

### 3.1 CaaS framework processes

In a nutshell, the CaaS framework will enable developers to create template workflows based on the functional components and correspondent semantic models so that they can be not only shared, but also executed later on by end-users or applications. The aim here is therefore to provide a set of tools and components to manage the lifecycle of the aforementioned templates, namely, Creation, Resolution and Execution:

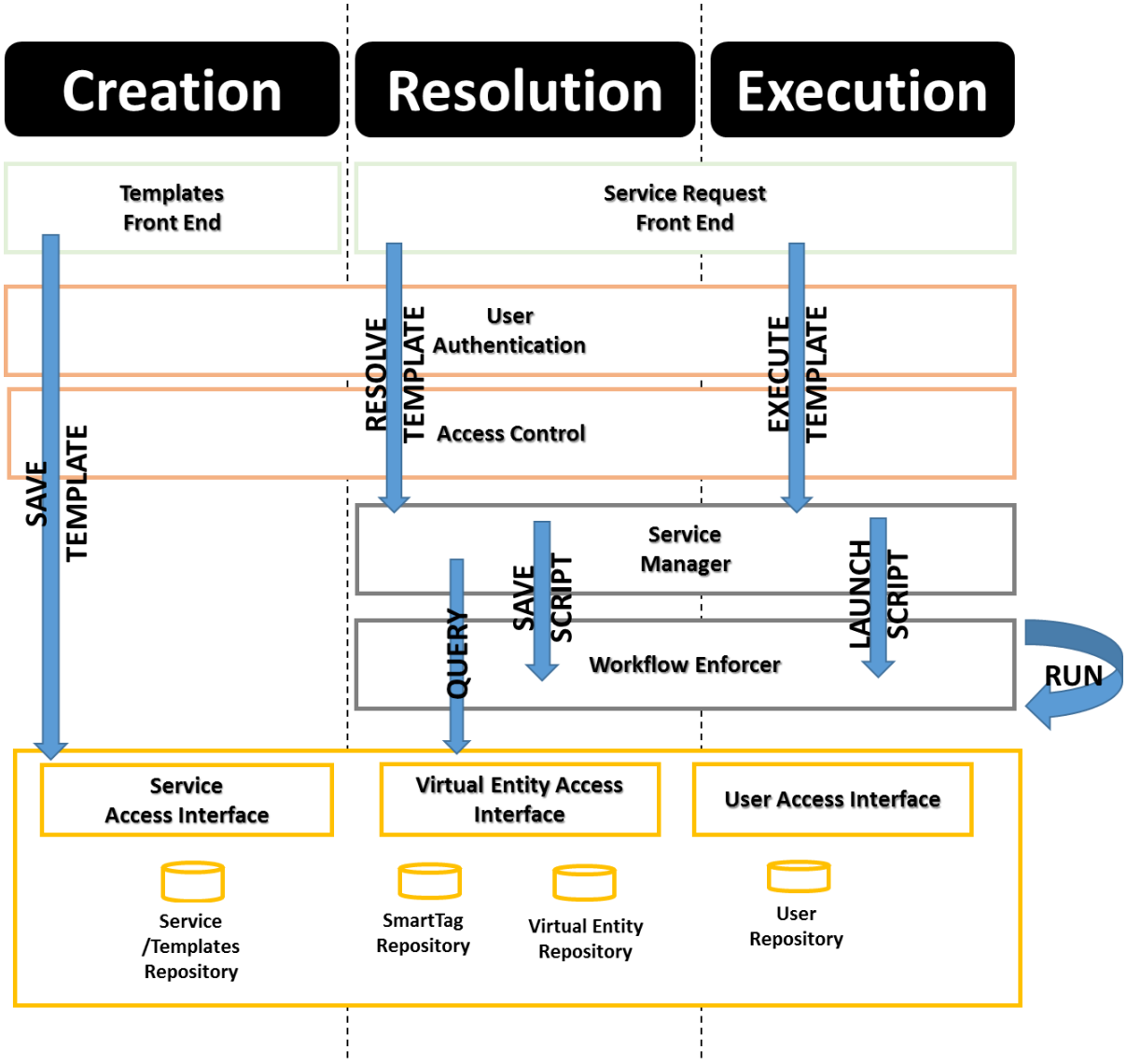


**Figure 2. Simple schematic of the CaaS processes**

### *3.1.1 Template Creation*

The first step is to create the template that describes the workflow in terms of data flows and components to execute a particular functionality.

The template creation will typically be done by a developer via the **Templates Front End.** This component facilitates the creation of the templates in a graphical way. It is not involved in the service composition execution process, but it is still a key component since it allows a developer to define the template and store it in the **Service Template Repository**.

For the **Template Front End** we are extending Node-RED, a well know and flexible workflow designer to produce the templates. The format of these templates is compliant with the Node-RED syntax, however a template will not be directly “hard-coded” and runnable, it will be

exported to the the **Service Template Repository** so that it can be resolved by the **Service Manager**.

### *3.1.2 Template Resolution*

The second step is to resolve the correspondent services in the template to actual implementations in one or different platforms. This goes along the lines of TagItSmart not being a dedicated fully-fledged platform, but a series of components that enable the integration of the SmartTags functionality into existing implementations, as long as these provide Web interfaces and are described accordingly.

The templates, as it will be shown later, guide the **Service Manager** to understand how to process the service requests coming from a user/application through the **Service Request Front End**, in terms of resolving the components that need to be orchestrated (e.g. which Virtual Entities and their associated services are involved, which platform services in the back-end, e.g. SmartTag Repository, Virtual Entity Repository, Service Repository, as well well as possible external services).

The **Service Manager** will use the information described in the template and the corresponding semantic models to discover in the correspondent repositories the services that are needed to execute the template. All these repositories are accessed through their corresponding interfaces (**Virtual Entity Access Interface and Service Access Interface**) so that the **Service Manager** can search for suitable entries in these repositories and -if found- configure them appropriately. In this process, user information is also accessed through the **User Access Interface**.

Finally, once suitable back-end services and services offered by VEs are found and configured, the **Service Manager** creates a script file that can be executed.

### *3.1.3 Template Execution*

The final step is to run and monitor the workflow. The resolved script will be sent to the **Workflow Enforcer**. This component is able to run the script and monitor step-by-step the service execution during runtime and act as the “middle man” between components, whenever this is required.

The **Workflow Enforcer** is therefore a workflow engine that understands the syntax of the script that is produced by the Service Manager. In the project we have integrated two underlying IoT Platforms with components that serve as the basis for the **Workflow Enforcer**. This led to the decision of using javascript and node.js as the preferred language, due to the relatively easy connection with both underlying platforms (DunavNET Platform, based on Microsoft Azure and EVRYTHNG Platform, based on Amazon Lambda)

## 3.2 CaaS framework example

The following example illustrates the steps among the components that are involved in the CaaS process:

### *3.2.1 Template Creation*

A developer logs into the **Templates Front End** and inserts, through the Service Access Interface, Service Templates in the Service Template Repository. As it will be shown in Section 4.1.1, these templates act as a guide so that a high level service request issued afterwards by a “non-technology aware” user of the platform, can be translated to low level service requirements.

In order to use the Templates Front End, a developer must log-in with username and password to the TagItSmart environment, as shown in Figure 3.

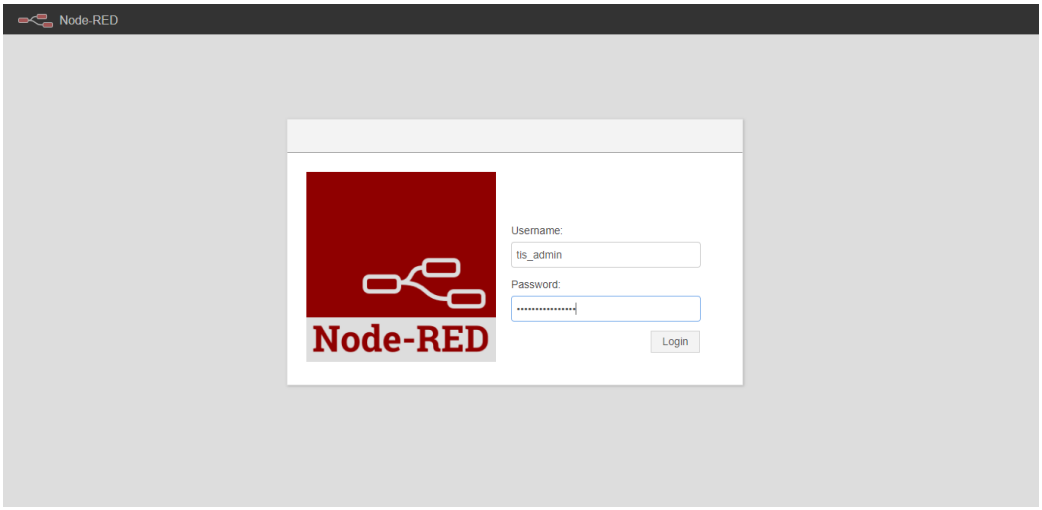


**Figure 3. Templates Front End Login**

Once logged in, a developer can create Service Templates by connecting together the inputs and outputs of the available nodes provided in the Node-RED environment.

As shown in Figure 4 on the left side under the TagItSmart group there are several available nodes that a developer can use to create a Service Template. These nodes provide functional components as described in the architecture and are not bind yet to an actual service. For example these are an Authentication node, an Encoding node, a VE Lifecycle Manager node, a Stream Processing node and a Notification node.

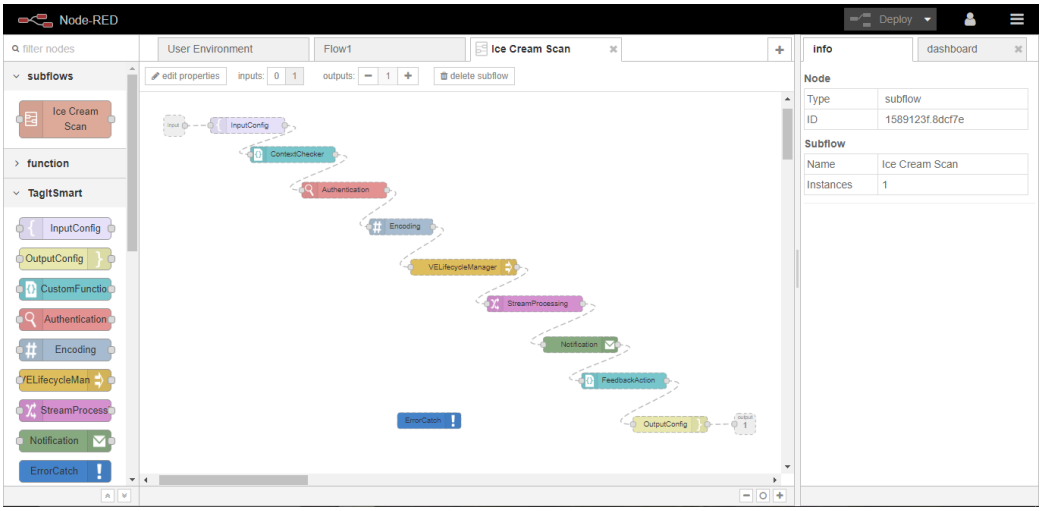


**Figure 4. Service Templates**

These nodes are completely customizable and are filled with the information needed that will be written in the Service Template Repository. Additionally there are an Input and an Output Configuration node which allows to customize the inputs and outputs of the service. Moreover it is possible to design Custom Function nodes which do not bind to an actual service, but implement the service logic in place within the flow. That is, it is also possible to pass some relatively basic functions to be executed locally by the Workflow Enforcer without having to rely on VEs or the back-end to provide these.

The resulting Service Template, which is another node in Node-RED, is exported to the Template Repository via an export entry in the menu which issues a request to the Service Access Interface, as shown in Figure 5.

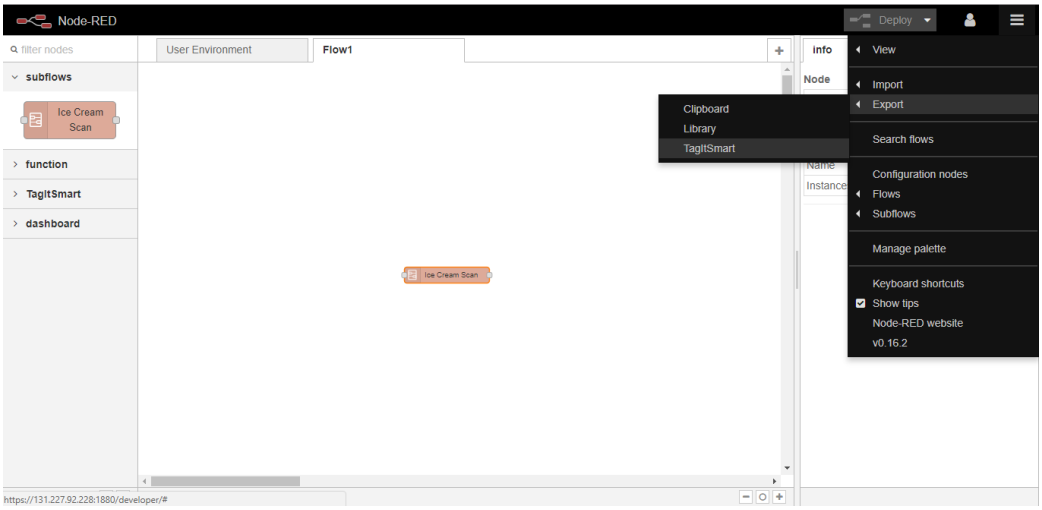


**Figure 5. Export to Service Template Repository**

This will make the template available in the Service Template Repository so that it can be afterwards used.

### *3.2.2 Template Resolution*

As mentioned before, templates need to be resolved to actual services in order to execute them. When a user of the platform issues a service request through the Service Request Front End, this is directed to the Service Manager.

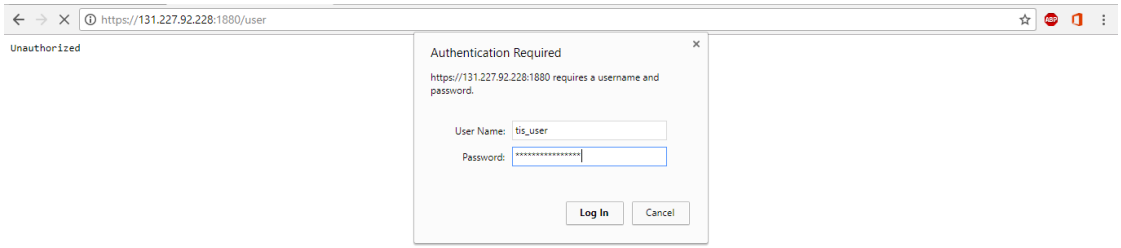


**Figure 6. Service Request Front End Authentication**

As shown in Figure 6, a user needs to log into the Service Request Front-End with his credentials which are checked by using the User Authentication/Access Control components through the User Access Interface against the credentials stored in the User Repository.

If the user is authorized to use the platform, the request sent to the Service Manager through the form of Figure 7 starts the relevant search through the Service Access Interface in the Service Template Repository, whether a Service Template exists that describes what the service request needs from the platform.

If such a Service Template exists, it is returned to the Service Manager. Such a template needs to be resolved, as it has “fields” that are at this stage incomplete and it is the role of the Service Manager to fill in the missing information. Service Request configuration parameters have to be provided as well before the Service Manager can resolve the template, which is used by the Service Manager for finding relevant VEs and Services.

Then the Service Manager through the available query functionality of the Virtual Entity Access Interface, Service Access Interface and User Access Interface attempts to retrieve auxiliary information from the SmartTag and User Repository and find suitable backend and VE services that are included as needed in the corresponding Service Template.

Once all such needed services are found, the Service Manager configures them appropriately both for operations and communication purposes (i.e. a specific setting in a printer and a

specific endpoint to which a service needs to forward its output once it performs its operations) and finally creates the workflow script to be executed.

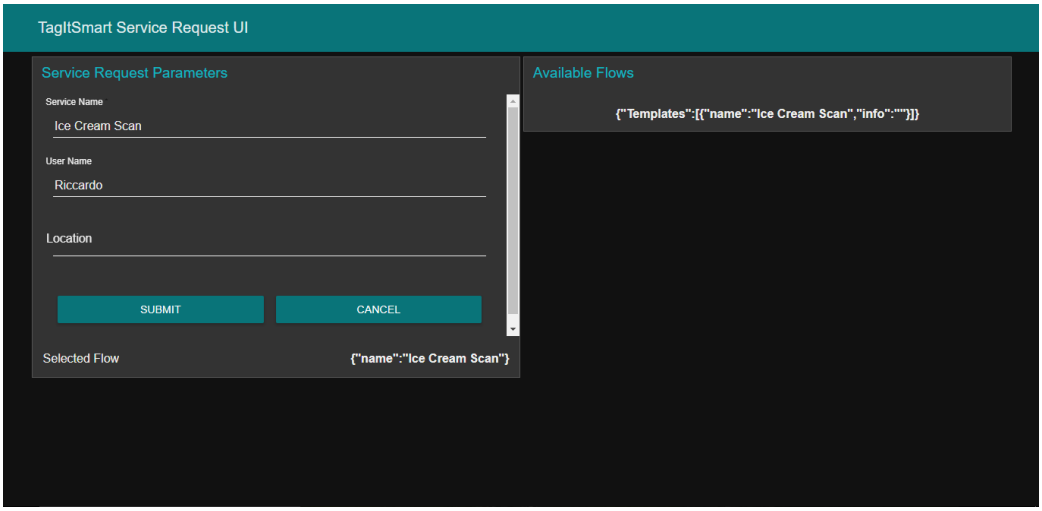


**Figure 7. Service Request Front End**

### *3.2.3 Template Execution*

The Service Manager passes to the Worklow Enforcer the now completed script (part of Service Template) which the Workflow Enforcer needs to execute as the “middle man” to make sure that all steps in the runtime service execution take place in the correct order and with correct information passing from one component to the other.

A user of the TagItSmart platform issues a composition request through the Service Request Front End, as shown in Figure 7. The Service Request requires a Service Name, which is selected among a list of available Service Templates names reported along in a list with their brief descriptions. Service Request input parameters have to be provided as well before the execution is launched.

# Section 4 - CaaS components

## 4.1 Repositories for exposing information for service composition

The repositories are the underlying components where the data related to the different entities described in the architecture are stored. They provide basically a database of some type, depending on the nature of the data stored and the capabilities needed. The following sections describe each one of the repositories that interact directly with the CaaS components.

### *4.1.1 Service Template Repository*

In TagItSmart, the Service Template Repository is in charge of storing all the necessary information needed by the Service Manager in order to effectively compose the required services needed to instantiate a particular flow. As mentioned in the previous sections, this template is the result of a placement of elementary building blocks or *nodes* in the Templates Front-End, which are mainly designed by the TagItSmart developers, but can further be customized in a flexible way within the Node-RED environment.

The Service Template Repository used is a NoSQL MongoDB [10] database which has been selected for its easiness to setup and because it uses a schema-free, document-based store which allows for deeply nested data models. These types of models match one on one the description of a particular flow with JSON object arrays and allow for easy and efficient parsing of the corresponding JSON objects. All the service templates are consequently stored as JSON objects within a single collection, in order to perform CRUD (Create, Read, Update, Delete) requests and queries in an efficient fashion.

**Figure 8. Template Insertion Process**

Figure 8 reports the template insertion process into the repository. As a consequence of an ``Export to TagItSmart'' action from the Templates Front-End, the customized template is then passed as a JSON object with an HTTP POST request to the Service Access Interface part pertaining to the Service Template Repository. This was implemented in Node.JS as an Express.JS web application framework with a particular Mongoose.JS schema which implements two keys:

- `"name":` a unique and required value of type String, which names the current service template.
- `"flow":` a JSON object representing the template exported from the Node-RED front-end.

Following a successful store in the MongoDB *templates* database and collection, the resulting JSON object which will be used by the Service Manager will therefore be composed by four main keys:

- "_id": an object id generated by the MongoDB database.
- "flow": a nested JSON array which provides the actual configuration of services and input/outputs.
- "name": a name for the service template, e.g. "Ice Cream Scan".
- "__v": a version key.

The template generated via Node-RED contains several JSON objects. These are an object for the configuration of the input and outputs and for the description of the service template, an object for the description of the UI related parameters such as the placement and unique identifier and finally, several objects each of them fully describing a node either representing a function or a service in its functions, identification, connectivity and UI placement parameters.

The first JSON object of the template, other than descriptive parameters such as the name of the template, an identifier and a brief description also provides the configuration parameters for the inputs and outputs nodes connectivity. The keys it provides are:

- "out": nodes connectivity parameters for the service template outputs.
- "in": nodes connectivity parameters for the service template inputs.
- "info": a short description of the service template functionality, editable from within the Templates Front End and to be displayed in the Service Request Front-End.
- "name": the unique name of the service template provided.
- "type": the type of object, as generated by Node-RED, i.e. "subflow".
- "id": the id of the service template, as generated by Node-RED.

While most of keys in this JSON object represent string values, the "in" and "out" keys are JSON arrays which contain both the UI related graphical location coordinates as "x" and "y" keys and the connectivity information as a "wires" JSON object.

The "wires" object contains the "id" keys of the nodes connected either to the input or output of the service template. Typically, by design the service template has a single input node, so there is only one "id" key in the "out" object referring to the last node in the template. Similarly, there is a single "id" key in the "in" object referring to the first node in the template. Please note that even though this single node connectivity for both input and output seems a limitation, this does not necessarily mean that the resulting flow has to be limited to one input and one output, because the input and output nodes could be catering for the needs of multiple input and outputs. Moreover, multiple service templates could be easily created within the Templates Front-End to serve different inputs and outputs needs.

After the first JSON object of the service template there are all the JSON objects related to the various services and functions designed in the Templates Front End. These include descriptions of services, such as an Authentication Service or a VE Lifecyle Manager Service as well as ad-hoc unannotated functions, which do not require semantic resolution by the Service Manager but actually implement in place service logic living inside the TagItSmart execution engine (see Figure 9). The Workflow Enforcer is indeed capable of executing ad-hoc customized javascript code without needing any call to services which are "external" from its point of view. Example thereof are either a ContextChecker function which provides the

capability to examine the context variables injected as the flow input or the instantiation of an IFTTT (if this then that) conditional statement following a particular service response.

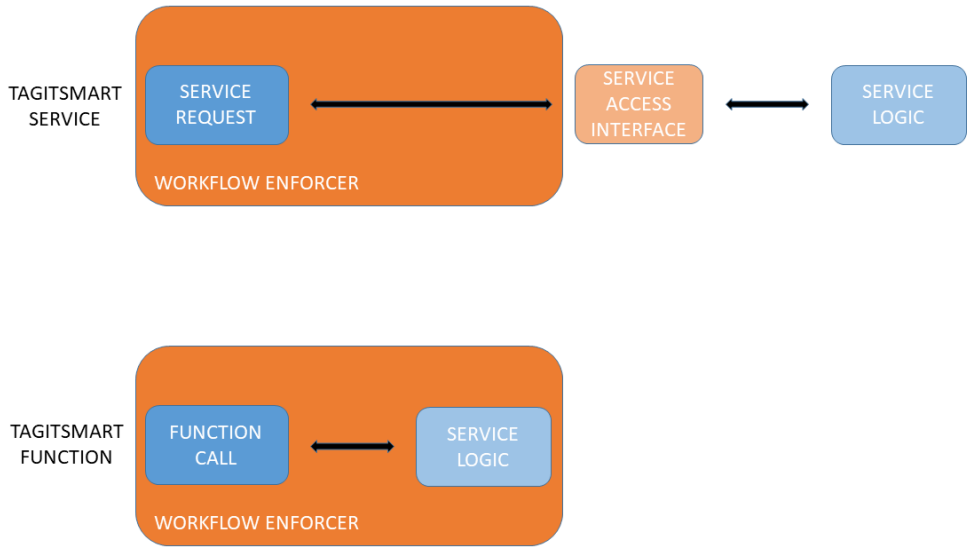


**Figure 9. Services and Functions Nodes**

Each one of these services and function are JSON objects with the common following keys which describe all the aspects of the nodes:

- `"wires":` a JSON object similar to the one described in the first JSON object of the template which contains the `"id"` key of the subsequent node in the service template.
- `"x":` the x coordinate of the node position in the Templates Front-End UI.
- `"y":` the y coordinate of the node position in the Templates Front-End UI.
- `"noerr":` a variable automatically generated by Node-RED which represents the error in the node.
- `"outputs":` the number of outputs of the node.
- `"func":` the javascript code template for the “to-be-resolved” service function call or function call to be run by the Workflow Enforcer, as well as their response or error handling.
- `"secfunc":`in case of a service node, the javascript code template for a service request to the corresponding service to be semantically resolved by the Service Manager in the necessary service parameters and run by the Workflow Enforcer; in case of a function node, the actual javascript code representing the service logic.
- `"name":` an optional name for the service or function.
- `"z":` the identifier of the service template to which this node belongs to, i.e. the one of the first JSON object in the service template aforementioned.
- `"type":` the type of node represented, which names the node category i.e. Authentication, Encoding, CustomFunction, etc…
- `"id":` the identifier of the node function or service.

Two special types of nodes are the first and last nodes in the service template, which are designed to be configuration nodes and therefore do not correspond to actual services or functions, but are needed for specifying the inputs and outputs of the flow and are used by the Service Manager at composition to configure the inputs and outputs.

Their structure as JSON objects is the same of the service or function nodes, meaning that they share the same keys, but do not provide a "secfunc" key. Moreover the "func" key represents a JSON object and not javascript code with some ad-hoc keys to be interpreted by the Service Manager.

The first node has a "type" field with value "InputConfig" and represents a JSON object with the following main keys:

- "name": a required name for the flow used by the Service Manager.
- "filter": the required filters for the types on the event action which triggers the flow execution.
- "context": the additional contextual parameters to be initialize at the beginning of the flow together with the "customFields" of the event action which triggers the flow.
- "dependencies": the javascript dependencies needed at execution time to be injected by the Service Manager into the Workflow Enforcer.

For example the InputConfig node JSON function for an "IceCream-Scan" flow is the following:

```
{
  "name": "IceCream-Scan",
  "filter": {
    "event.action.type":"_smartTagRead"
  },
  "context": {
    "allowedProductID": "U3gECAt8BDsRQKRRwDAA8t2n"
  },
  "dependencies": {
    "evrythng-extended": "^4.1.0",
    "request": "*",
    "request-promise": "*"
  }
}
```

The last node in the service template has a "type" field with value "OutputConfig" and represents a JSON object with the following main keys:

- "name": a required name for the flow used by the Service Manager.
- "filter": the required filters for the types on the event (feedback) action which needs to be generated by the Workflow Enforcer and that ends the flow execution.

For example the OutputConfig node JSON function for an "IceCream-Scan" flow is the following:

```
{
  "name": "IceCream-Scan",
  "filter": {
    "event.action.type":"_smartTagReadDone"
```

```
    }
}
```

Within the service template there is also a node which has a "type" field with value "ErrorCatch" which is needed by the Service Manager, but doesn't represent any real service or function. This node has an empty "wires" section since it is tipically not connected to any other node and has the purpose of configuring the flow output in case of an error throw in any of the services or functions

Finally, the last JSON object in the service template describes the single node representing the entire service template, which is exported from the Templates Front-End. Since in principle this object is also a node within Node-RED it allows the following keys (and in principle nesting of service templates) :

- "wires": a JSON object similar to the one described above which describes the eventual connectivity of the Service Template.
- "x": the x coordinate of the node position in the Templates Front-End UI.
- "y": the y coordinate of the node position in the Templates Front-End UI.
- "z": the identifier of the service template to which this node belongs to, i.e. the nested template.
- "type": the type of node represented, which names the node category, i.e. "subflow:id"
- "id": the identifier of the node.

An example of one of the flow stored within the Service Template Repository for the scan of an Ice Cream product is reported below. This includes a ContextChecker function, an Authentication Service, a Decoding Service, a VE Lifecycle Manager Service, a Stream Processing Service, a Notification Service and a FeedbackAction function.

---

```
{
  "_id" : ObjectId("5984456aa8d7d61585230a67"),
  "flow" : [
    {
      "out" : [
        {
          "wires" : [
            {
              "port" : 0,
              "id" : "faf45dc3.e5a6c"
            }
          ],
          "y" : 682.333381652832,
          "x" : 1197.6667547225952
        }
      ],
      "in" : [
        {
          "wires" : [
            {
              "id" : "cec6e82a.0bac1"
```

```
        }
      ],
      "y" : 120.66665458679199,
      "x" : 122.66668128967285
    }
  ],
  "info" : "",
  "name" : "Ice Cream Scan",
  "type" : "subflow",
  "id" : "1589123f.8dcf7e"
},
{
  "wires" : [
    [
      "e6f1806c.0b6698"
    ]
  ],
  "y" : 119.99998474121094,
  "x" : 260.83333587646484,
  "noerr" : 0,
  "outputs" : 1,
  "func" : "{\n  \"name\": \"IceCream-Scan\",\n  \"filter\": {\n
  \"event.action.type\":\" smartTagRead\"\n  },\n  \"context\":
  {\n    \"allowedProductID\": \"U3gECAt8BDsRQKRRwDAA8t2n\"\n
  },\n  \"dependencies\": {\n    \"evrythng-extended\":
  \"^4.1.0\",\n    \"request\": \"*\",\n    \"request-promise\":
  \"*\"\n  }\n}",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "InputConfig",
  "id" : "cec6e82a.0bac1"
},
{
  "wires" : [
    [ ]
  ],
  "y" : 682.9999847412109,
  "x" : 1056.8333358764648,
  "noerr" : 0,
  "outputs" : 1,
  "func" : "{\n  \"name\": \"IceCream-Scan\",\n  \"filter\": {\n
  \"event.action.type\":\"_smartTagReadDone\"\n  }\n}",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "OutputConfig",
  "id" : "faf45dc3.e5a6c"
},
{
  "wires" : [
    [
      "1687c905.c461af"
```

```
      ]
    ],
    "y" : 241.99998474121094,
    "x" : 442.83333587646484,
    "noerr" : 0,
    "outputs" : 1,
    "secfunc" : "var request = require('request-promise');\nvar
    options = {\n    method: 'POST',\n    uri: 'url-to-be-
    resolved',\n    body: context.credentials,\n    json: true, //
    Automatically stringifies the body to JSON\n    headers: {\n
    'Content-Type': 'application/json',\n        'Authorization':
    context.appApiKey\n    }\n};\nreturn request(options);",
    "func" : "// Authentication
    Service\nlogger.info('Authentication!');\nreturn
    tagitsmartService.Authentication.validate(context)\n
    .then(function (result) {\n        logger.info('Authentication
    Successful!');\n        context.userApiKey =
    result.evrythngApiKey;\n    }, function (error) {\n
    logger.info('Authentication Error!');\n        throw error;\n
    });\n// Authentication Service",
    "name" : "",
    "z" : "1589123f.8dcf7e",
    "type" : "Authentication",
    "id" : "efc7d010.f9b77"
  },
  {
    "wires" : [
      [
        "57e60b31.846554"
      ]
    ],
    "y" : 310.99998474121094,
    "x" : 549.8333358764648,
    "noerr" : 0,
    "outputs" : 1,
    "secfunc" : "var request = require('request-promise');\nvar
    options = {\n    method: 'POST',\n    uri: 'url-to-be-resolved'
    + '?filter=method=2d,ir&type=qr_code,image',\n    body: {
    image: context.image },\n    json: true, // Automatically
    stringifies the body to JSON\n    headers: {\n        'Content-
    Type': 'application/json',\n        'Authorization':
    context.appApiKey\n    }\n};\nreturn request(options);",
    "func" : "// Encoding Service\nreturn
    tagitsmartService.Encoding.decode(context)\n    .then(function
    (results) {\n        //Decoding successful\n
    logger.info('Decoding Successful!');\n        context.product =
    results[0].results[0].product;\n
    logger.info(JSON.stringify(context.product, null, 4));\n
    logger.info(JSON.stringify(results, null, 4));\n\n        if
    (!context.updates) {\n            context.updates = [];\n
    }\n\n        //update values\n        for (let result of
```

```
  results) {\n            if (result.meta.method === 'ir') {\n
  context.updates.push({\n                    key:
  'consumption_temperature_ok',\n                    value:
  true\n                });\n            }\n            else if
  (result.meta.method === '2d') {\n
  context.qrValue = result.meta.value.slice(-2);\n\n
  return app.product().read({\n                    params: {\n
  filter: 'tags=' + context.qrValue\n                    }\n
  }).then(function (product) {\n
  context.product = product[0];\n
  logger.info('Product Found by Identifier');\n\n
  // Check product is ice cream\n                    if
  (context.product.id !== context.allowedProductID){\n
  throw new Error('Unrecognized Type of Product');\n
  }\n                    // Checked product is ice cream\n
  context.updates.push({\n                        key:
  'cold_chain_broken',\n                        value:
  !(context.product.customFields[context.qrValue.toString()] ===
  'cold_chain_ok')\n                    });\n
  });\n            }\n        }\n    }, function (error) {\n
  //Decoding error\n        logger.info('Decoding Error!');\n
  throw error;\n    });\n// Encoding Service",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "Encoding",
  "id" : "1687c905.c461af"
},
{
  "wires" : [
    [
      "b7c3894e.b3502"
    ]
  ],
  "y" : 455.99998474121094,
  "x" : 801.8333358764648,
  "noerr" : 0,
  "outputs" : 1,
  "secfunc" : "var request = require('request-promise');\nvar
  options = {\n    url: 'url-to-be-resolved',\n    method:
  'POST',\n    json: true,\n    rejectUnauthorized: false,\n
  requestCert: true,\n    agent: false,\n    body: { \n
  arg1: context.cold_chain_broken,  \n        arg2:
  context.consumption_temperature_ok\n    }\n};\nreturn
  request(options);",
  "func" : "//Stream Processing Service\nreturn
  tagitsmartService.StreamProcessing.process(context)\n
  .then(function (result) {\n        logger.info('Stream
  Processing successful with output!' +
  JSON.stringify(result.output1));\n        context.message =
  result.output1;\n    }, function (error) {\n
```

```
  logger.info('Stream Processing Error!');\n        throw
  error;\n    });\n//Stream Processing Service",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "StreamProcessing",
  "id" : "930c5382.8de11"
},
{
  "wires" : [
    [
      "1b9208a3.20ac27"
    ]
  ],
  "y" : 532.9999847412109,
  "x" : 887.8333358764648,
  "noerr" : 0,
  "outputs" : 1,
  "secfunc" : "var request = require('request-promise');\nvar
  options = {\n    method: 'POST',\n    uri: 'url-to-be-
  resolved',\n    form: {sku: context.message},\n};\nreturn
  request(options);",
  "func" : "//Notification Service\nreturn
  tagitsmartService.Notification.notify(context)\n
  .then(function (result) {\n        //Notification successful\n
  logger.info('Notification to Screenly Sent Successfully!');\n
  }, function (error) {\n        logger.info('Notification to
  Screenly Error!');\n        throw error;\n
  });\n//Notification Service",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "Notification",
  "id" : "b7c3894e.b3502"
},
{
  "wires" : [
    [
      "930c5382.8de11"
    ]
  ],
  "y" : 386.99998474121094,
  "x" : 667.8333358764648,
  "noerr" : 0,
  "outputs" : 1,
  "secfunc" : "var request = require('request-promise');\nvar
  options = {\n    method: 'PUT',\n    uri: 'url-to-be-resolved'
  + context.product.id + '/properties',\n    body:
  context.updates,\n    json: true, // Automatically stringifies
  the body to JSON\n    headers: {\n        'Content-Type':
  'application/json',\n        'Authorization':
  context.userApiKey\n    }\n};\nreturn request(options);",
```

```
  "func" : "//VELifecycleManager Service\nreturn
  tagitsmartService.VELifecycleManager.update(context)\n
  .then(function (result) {\n        //Update successful\n
  logger.info('VE Update Successful!');\n
  logger.info(JSON.stringify(result, null, 4));\n\n
  context[result[0].key] = result[0].value;\n
  context[result[1].key] = result[1].value;\n\n    }, function
  (error) {\n        logger.info('VE Update Error!');\n
  throw error;\n    });\n//VELifecycleManager Service",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "VELifecycleManager",
  "id" : "57e60b31.846554"
},
{
  "wires" : [ ],
  "y" : 675.9999847412109,
  "x" : 602.8333358764648,
  "noerr" : 0,
  "outputs" : 0,
  "func" : " logger.error(\"Error executing the flow \" +
  JSON.stringify(error, null, 4));\n\n// Defined in Node-
  RED\ncontext.workflowOutput = {\n    product:
  context.product.id,\n    customFields: {\n        error:
  error\n    }\n};\n// Defined in Node-RED",
  "name" : "",
  "z" : "1589123f.8dcf7e",
  "type" : "ErrorCatch",
  "id" : "6d1442b6.1b77ec"
},
{
  "wires" : [
    [
      "efc7d010.f9b77"
    ]
  ],
  "y" : 174.99998474121094,
  "x" : 345.83333587646484,
  "noerr" : 0,
  "outputs" : 1,
  "secfunc" : "//This function could define any transformations
  or checks needed in the context\n// e.g. check that all the
  inputs are available, combine EVT info in the context\nreturn
  new Promise(function (resolve, reject) {\n    if (context !==
  null) {\n        resolve(context);\n    } else {\n
  reject(\"Context not valid\");\n    }\n})",
  "func" : "tagitsmartFunction.ContextChecker.check(context)\n",
  "name" : "ContextChecker",
  "z" : "1589123f.8dcf7e",
  "type" : "CustomFunction",
  "id" : "e6f1806c.0b6698"
```

```
    },
    {
      "wires" : [
        [
          "faf45dc3.e5a6c"
        ]
      ],
      "y" : 600.9999847412109,
      "x" : 969.8333358764648,
      "noerr" : 0,
      "outputs" : 1,
      "secfunc" : "//This function could define any transformations
      or checks needed in the context\n// e.g. generate the output of
      the flow\ncontext.workflowOutput = {\n    product:
      context.product.id,\n    customFields: {\n        product:
      context.product\n
      }\n};\n\nlogger.info(JSON.stringify(context.workflowOutput));\n
      \nreturn new Promise(function (resolve, reject) {\n    if
      (context.workflowOutput !== null) {\n
      resolve(context);\n    } else {\n        reject(\"Workflow
      output not valid\");\n    }\n});",
      "func" : "return
      tagitsmartFunction.FeedbackAction.configure(context)",
      "name" : "FeedbackAction",
      "z" : "1589123f.8dcf7e",
      "type" : "CustomFunction",
      "id" : "1b9208a3.20ac27"
    },
    {
      "wires" : [
        [ ]
      ],
      "y" : 419,
      "x" : 538.5,
      "z" : "8d1c3778.84aed",
      "type" : "subflow:1589123f.8dcf7e",
      "id" : "2bc24fb5.c93fc8"
    }
  ],
  "name" : "Ice Cream Scan",
  "__v" : 0
}
```


---

### *4.1.2 Service Repository*

The Service Repository is responsible for making available the semantic descriptions of the services that are registered in the platform. Service descriptions are RDF [11] based representations of the services following the models used in TagItSmart (see D4.1 [12]). The implementation of the semantic repository, for the Service Repository and all the other

repositories in the remainder of this section, is based on a standard Triple Store (Virtuoso [13]) that is used for registration and query of the services through an SPARQL [14] endpoint.

The rest are some examples of services stored in the service repository; these are services offered both by the backend but also through VEs.

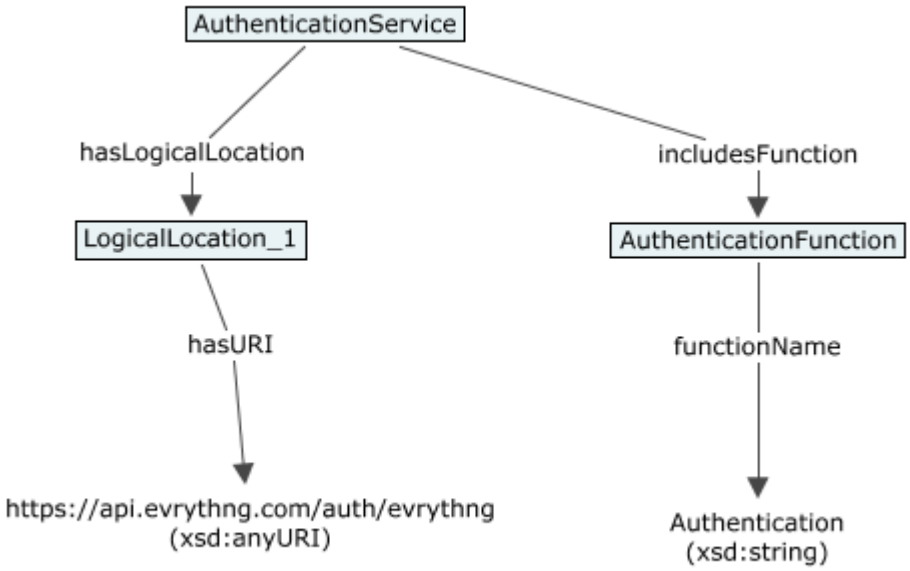


**Figure 10. Authentication service representation in the Service Repository**

Figure 10 shows an example of an Authentication service residing at the backend of the TagItSmart platform; this can be used to authenticate users based on their stored credentials. The representation allows for the endpoint where the service offering the Authentication function can be accessed, to be discovered through the LogicalLocation property.

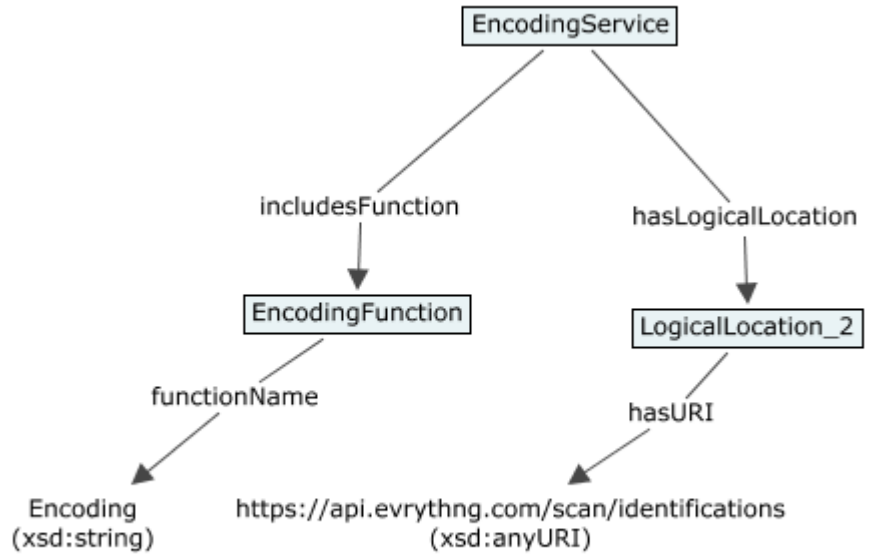


**Figure 11. Encoding service representation in the Service Repository**

Figure 11 shows an example of an encoding service residing at the backend of the TagItSmart platform; this can be used to transform raw SmartTag readings to meaningful readings. The representation allows for the endpoint where the service offering the encoding function can be accessed, to be discovered through the LogicalLocation property.

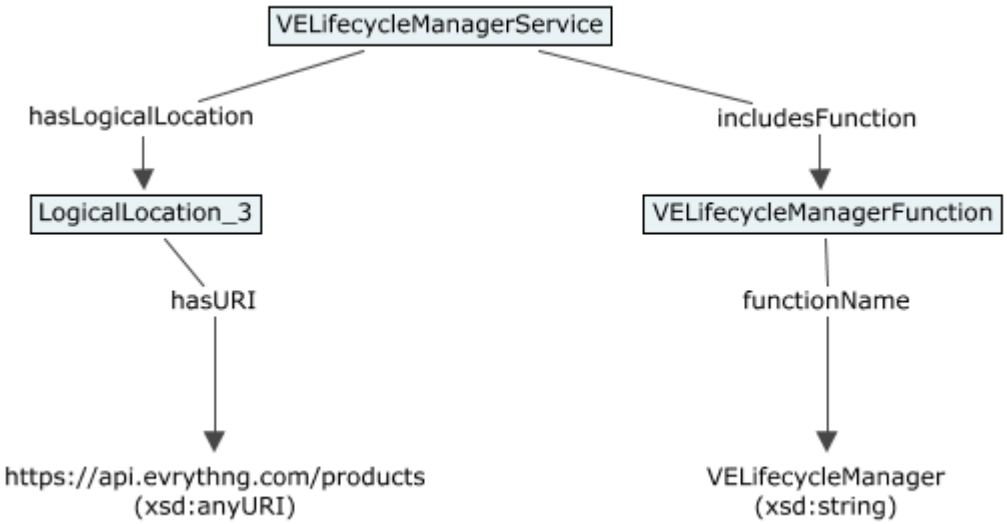


**Figure 12. VE lifecycle manager service representation in the Service Repository**

Figure 12 shows an example of a VE Lifeccycle Manager service residing at the backend of the TagItSmart platform; this can be used to update the status of a VE based e.g. on the value captured by a SmartTag reading. The representation allows for the endpoint where the service offering the VELifecycleManager function can be accessed, to be discovered through the LogicalLocation property.

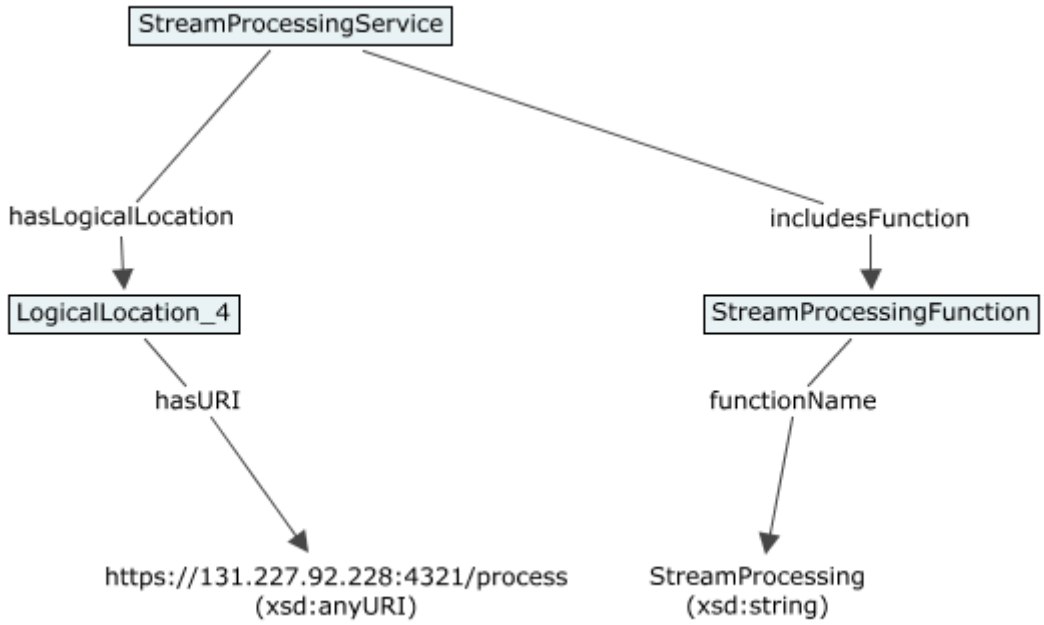


**Figure 13. Stream Processing service representation in the Service Repository**

Figure 13 shows an example of a stream processing service residing at the backend of the TagItSmart platform; this can be used to process the value captured by a SmartTag reading and perform decision making based on it. The representation allows for the endpoint where the service offering the StreamProcessing function can be accessed, to be discovered through the LogicalLocation property.

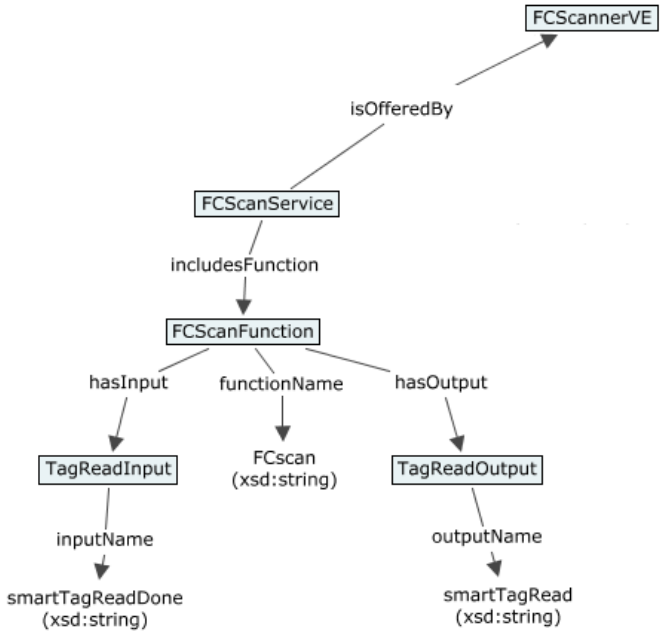


**Figure 14. FC scan service service representation in the Service Repository**

Figure 14 shows an example of an FC scan service offered by an FC scanner VE; this can be used to scan a SmartTag. The representation allows to represent the inputs/outputs of this service so that only suitable FC scan services are selected based on their properties and capabilities. As this service is offered by a VE, it is also linked to a VE instance (shown in the next section) and to a VE owner.

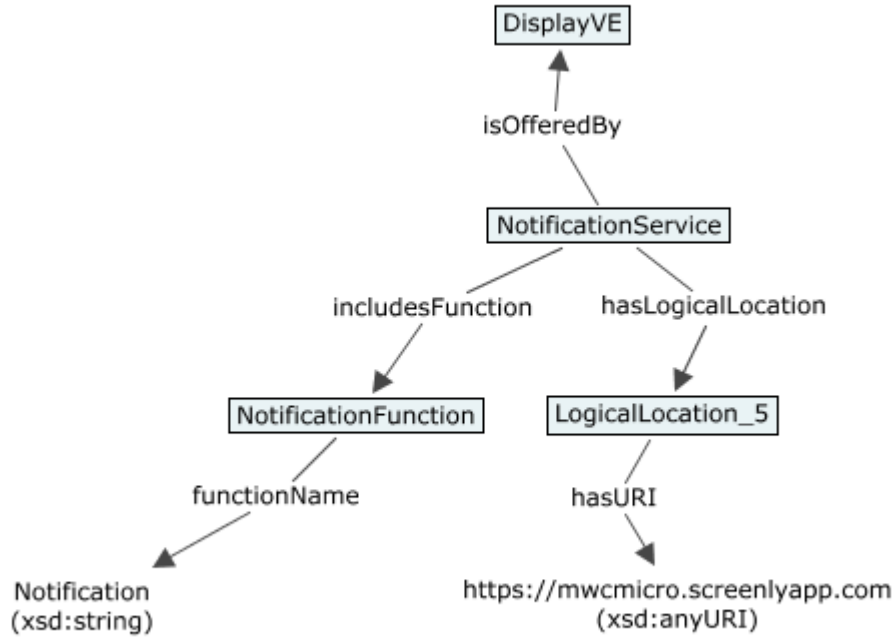


**Figure 15. Notification service representation in the Service Repository**

Figure 15 shows an example of a notification service offered by a display VE; this can be used to notify visually about the outcome of the SmartTag scan based on the conditions of the specific product with the SmartTag attached. The representation allows for the endpoint where the service offering the Notification function can be accessed, to be discovered through the LogicalLocation property. As this service is offered by a VE, it is also linked to a VE instance (shown in the next section) and to a VE owner.

### *4.1.3 VE Repository*

The VE Repository is responsible for storing the information about the VEs in the system. The VEs aggregate and link different types of information about the physical entity they represent. A VE has a semantic description associated, as well as a set of properties that are linked to the physical entity. These properties can be static or seldom updated, like for example the name, or they can be linked to dynamic properties that might change at a higher frequency and that also need to keep their history values. The VE repository takes care of the former type of properties by the use of a semantic repository.

The following are three VE representations of VEs stored in the VE repository corresponding to a Mobile device acting as an FC scanner VE, a Rasberri Pi screen acting as a display VE and also an Ice Cream VE (it is worth reminding that an item with a SmartTag attached is also considered a VE, since the SmartTag offers sensing capabilities).

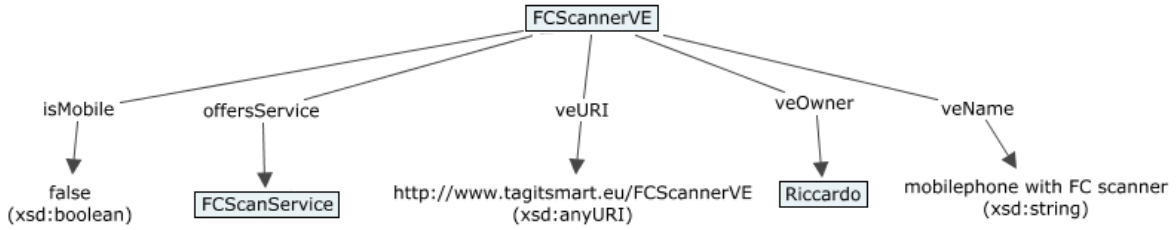


**Figure 16. FC scanner VE representation in the VE Repository**

Figure 16 shows an example of a mobile phone offering an FC scan service (see Figure 14) with owner Riccardo. Together with the FC scan service instance, it allows for querying and resolving whether a user Riccardo has a device that can act as an FC scanner.

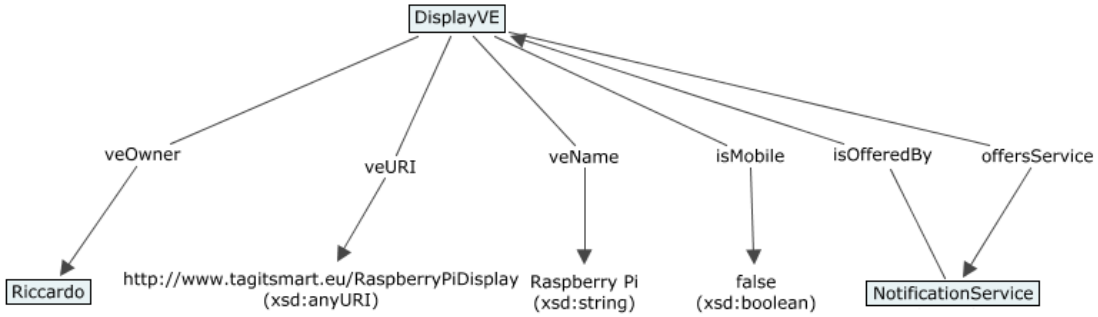


**Figure 17. Display VE representation in the VE Repository**

Figure 17 shows an example of a Rasberry Pi screen offering an notification service (see Figure 15) with owner Riccardo. Together with the notification service instance, it allows for querying and resolving whether a user Riccardo owns a device with notification capabilities and -if so- what is the endpoint whre this can be reached so it is triggered to display the appropriate notification.

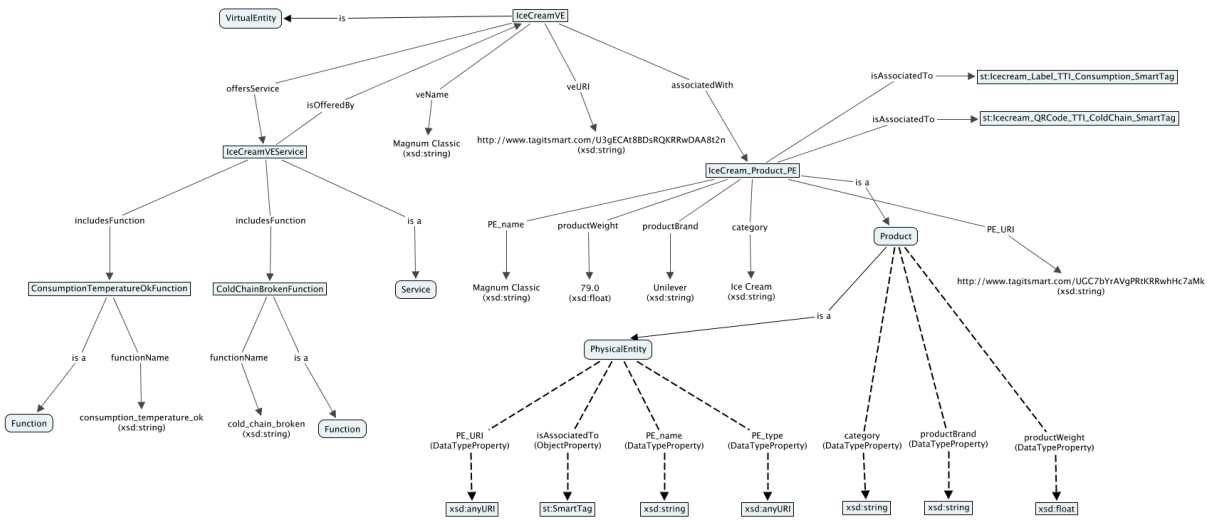


**Figure 18. Ice Cream VE and Ice Cream PE representation**

Figure 18 shows an example of how the VE, the PE (Physical Entity) and the SmartTags are connected. It is based on one of our use case examples, an Ice Cream PE, a product that has a SmartTag to sensor the state of the cold chain and the ideal consumption temperature. The Ice Cream VE provides the service to access the values that are sensed by the SmartTag. The definition of the SmartTag is shown later in section 4.1.5.

### *4.1.4 User Repository*

The User Repository holds the information about the users (these can e.g. be developers using the Templates Front End, users issuing requests through the Service Request Front End as well as FC-scanner users) and eventually the access control and permissions mechanism to access the rest of the components (to be defined as part of WP2/3 work). Similarly to the rest of the entities the user has a semantic description and the corresponding mapping to the underlying user management scheme.

The following are two examples of such entries in the User Repository, corresponding to a platform and an FC scanner user respectively.

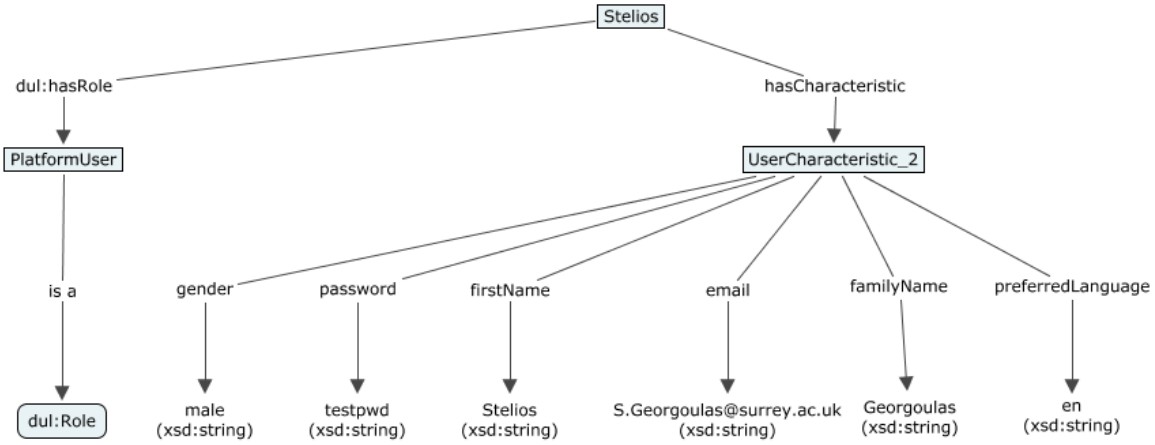


**Figure 19. Platform user representation in the User Repository**

Figure 19 shows an example of a user with name Stelios assigned a role as a platform user. Through the characteristic properties, his credentials can be queried and be used for access control purposes.

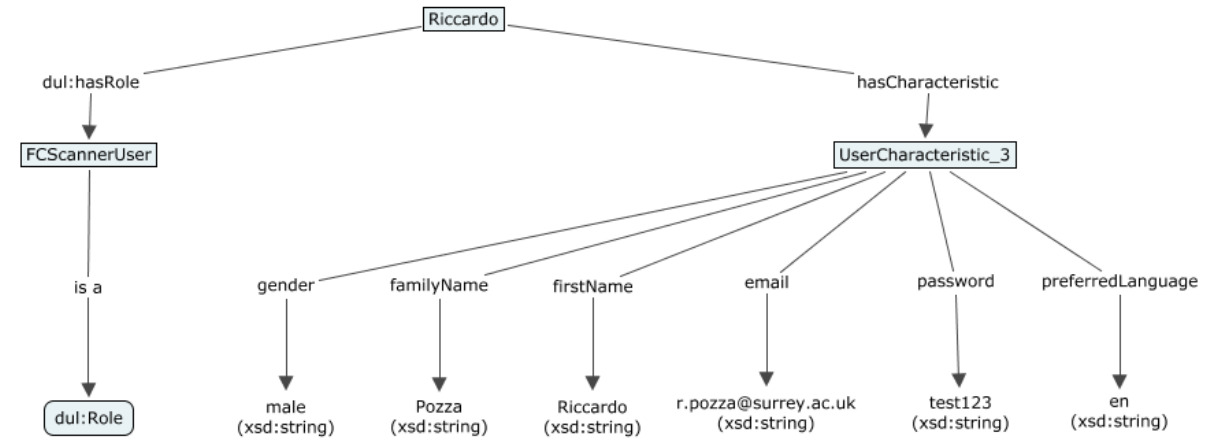


**Figure 20. FC scanner user representation in the User Repository.**

Figure 20 shows an example of a user with name Riccardo assigned a role as an FC csanner user. Through the characteristic properties, his credentials can be queried and used for access control purposes. In this case, user Riccardo based on the assigned role is not allowed to use the Front End to issue a service request but is allowed to send a SmartTag read to the platform and use the platform services for its subsequent processing.

### *4.1.5 SmartTag Repository*

SmartTags are a key entity in TagItSmart. They provide not only identification but also sensoring capabilities to the Physical Entities to which they are attached. The SmartTag repository is connected logically to the VE Repository, as a SmartTag plus the correspondent Physical Entity is what defines the VE.

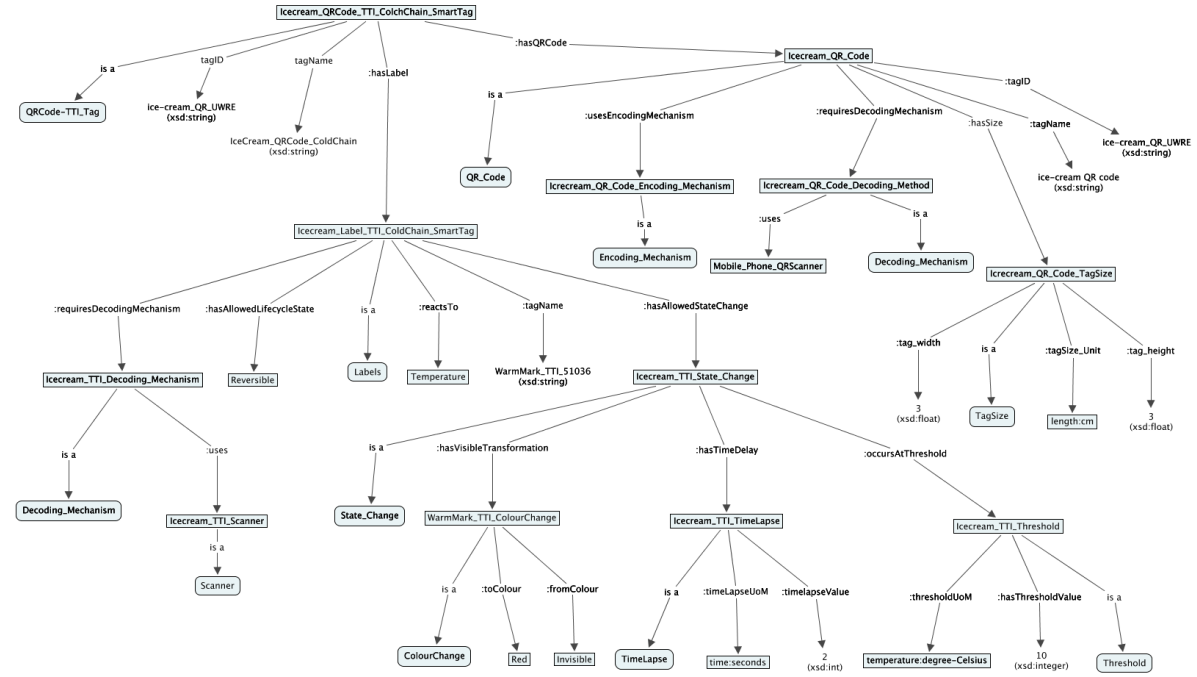


**Figure 21. Ice-cream SmartTag representation in the SmartTag Repository**

Figure 21 shows part of the definition of a SmartTag that follows the example of the ice cream used in one of the use cases. The SmartTag defined here is QR Code based and it is used to detect if the cold chain was broken. The SmartTag defines the QR part with its correspondent size and encoding/decoding mechanisms. The label describes the functional ink part, time, temperature threshold, state changes, etc.

## 4.2 Service setup and runtime execution components

### *4.2.1 Service Manager*

The Service Manager is a backend service developed in Node.js which is in charge of the managing composition requests originating the Service Request Front End. A composition request can be initiated by a user of the TagItSmart platform through the Service Request Front-End, as it was shown in Figure 7.

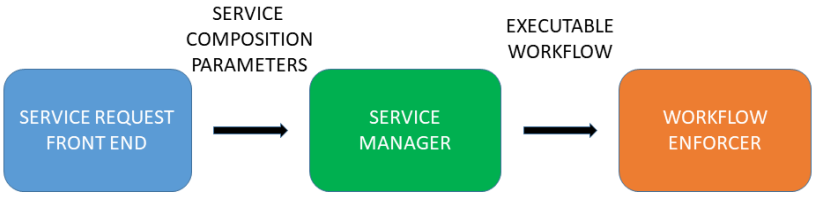


**Figure 22. Service Composition**

As described in the previous sections, a user logged into the Service Request Front-End is presented with a list of the service templates available in the Service Template Repository along with a short description of the type of service composition they implement. These are service templates developed in the Templates Front-End and implement complex functionalities of TagItSmart, such as SmartTag creation and printing.

Along with the list of available service templates, the user is presented with a web form where he can specify both mandatory and optional service parameters. These include the mandatory service template name along with optional user name and geolocation information (i.e. geofencing). For example, a user “A” logged into the platform could be authorized to generate a flow for scanning Smart Tags for either users “A” or “B”, as well as being able to generate a flow for printing a Smart Tag which needs to find an available a Printer VE closer or within a certain geographical location.

The Service Manager listens for HTTP POST requests with JSON body containing the following keys:

- `"template":` a mandatory name for the service template which needs to be matched against the available in the Service Template Repository.
- `"name":` an optional name of a user of the flow to be instantiated in order for them to use a particular flow.
- `"location":` an optional string containing geographical location information to be matched against the VEs.

By making optional the name and location it is possible to instantiate services which allow to be open to all users if needed or to any geographical location.

After receiving the service request, the Service Manager will query the Service Access Interface in order to retrieve the Service Template from the Service Template Repository as it can be seen in Figure 23.

**Figure 23. Service Template Match**

The service template name is searched within the database with a regular expression matching which looks for a partial string which it is not case sensitive and replaces eventual multiple white spaces with a single space. If the Service Manager can not find any service template matching the expression in the Service Template Repository, it returns an error message response. If instead it finds at least one service matching it returns the service template name of the first on the list of services found matching the expression.

Following the service template retrieval, the Service Manager starts by first checking that the user provided in the Service Request Front-End is actually a user of the TagItSmart platform.

In order to do so, and as it will be discussed further on in this section (see Figure 24), the Service Manager needs to check if a user exists with the provided username, by accessing the User Access Interface and the User Repository. In particular the Service Manager looks for an entry which has a specific mailbox username (*hasCharacteristic*) within the *email* field, e.g. userA for userA@example.com (see model in Section 4.1.4).

The Service Manager then parses the first JSON object in the template (see section 4.1.1 for details) and annotating the identification number of the first and last nodes in the service template.

The Service Manager then prepares on its filesystem a series of folders needed for annotating the workflow which results from the service composition task. In particular, it firstly cleans up eventually present folders and then creates the main directory (named `"workflow"`) for the workflow and two subdirectories: one for the semantically resolved services (named `"resolved-services"`) and one for the included functions (named `"included-functions"`).

The Service Manager then loops on the service template looking for a node of type `"InputConfig"` and parses the JSON contained within the node template key `"func"`, as well as annotating the identifier of the next node in the service template.

After finding the input configuration, the Service Manager loops within the service template looking for the next node identifier. Based on whether or not the first actual node representing either a service or a function was found, the Service Manager encapsulate the `"func"` function in a Node.JS Promise prototype of type `".then( function() {…} )"` to set the execution ordering of the various functions. This allows to avoid the well-know *"callback hell"* by returning function callback promises instead. This encapsulated main function is then appended to a variable which represents the `"main.js"` script which is executed by the Workflow Enforcer.

Then, the Service Manager searches within the node `"func"` function for either a `"tagitsmartFunction"` or a `"tagitsmartService"` keyword in order to differentiate between remote services and functions implementing the actual service logic within the Workflow Enforcer. For both cases, the Service Manager parses a name and a scope variable. These variables come from the definition of the actual services or function and are in the form of:

- `tagitsmartFunction.FunctionScope.FunctionName(context).`
- `tagitsmartService.ServiceScope.ServiceName(context).`

For example an Authentication Service request for validation of an username and password contained in a context variable could be in the form of:

- `tagitsmartService.Authentication.validate(context).`

If the node `"func"` function is a Function, the Service Manager then prepares the related entries in an `"index.js"` script to be stored into the `"included-functions"` folder. For each function, this is in the form of:

- `var FunctionScope = require('./FunctionScope.js');`
- `exports.FunctionScope = FunctionScope;`

Similarly, if the node "func" function is a Service, the Service Manager then prepares the related entries in an "index.js" script to be stored into the "resolved-services" folder. For each service, this is in the form of:

- `var ServiceScope = require('./ServiceScope.js');`
- `exports.ServiceScope = ServiceScope;`

In case of a Function, there is no need for semantic resolution so the node "secfunc" secondary function of the service template is encapsulated in a `exports.FunctionName = function(context) { … }` wrapper and written into the filesystem in the "workflow/included-functions" directory with filename "FunctionScope.js".

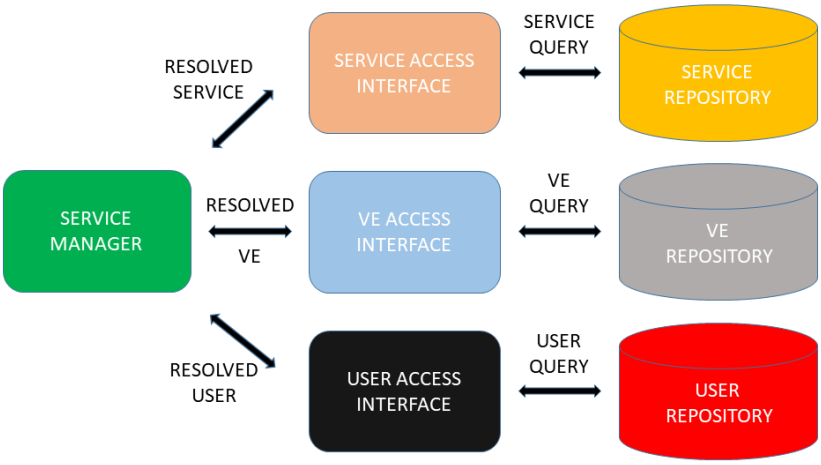


**Figure 24. Semantic Resolution of Services, VEs and Users**

In case of a Service, the Service Manager still encapsulates the node "secfunc" secondary function of the service template is in a `exports.ServiceName = function(context) { … }` wrapper and written into the filesystem in the "workflow/resolved-services" directory with filename "ServiceScope.js". However, as it can be seen from Figure 23, beforehand it also queries the Service Access Interface for the existance of platform services in the Service Repository.

In particular, for each platform service the Service Manager looks for the availability in the Service Repository of a service which has a function (*includesFunction*) which has a function name (*hasName*) matching the "ServiceScope". Moreover, if the service is intended to be offered by (*isOfferedBy*) a Virtual Entity, e.g. a Notification Service offered by a DisplayVE, then the Service Manager queries through the VE Access Interface for the availability of a VE that matches the owner user (*hasOwner*) specified in the Service Request Front End and for which the existance was checked at the beginning of the service composition task (see beginning of this section).

The Service Manager therefore resolves the missing parameters that are necessary for the actual service request (type of query, endpoints and data format) and stores these parameters in the placeholder provided within the node `"secfunc"` secondary function of the service template. For example, once resolved the HTTP URL of the service http://www.example.com/ it replaces the `"url"` key associated value placeholder in the `"options"` JSON object of the `"secfunc"` secondary function, thus obtaining `"var options = { url: 'http://www.example.com', … }"`.

The Service Manager then annotates the next services or function identifiers by parsing the `"wires"` object within the node `"func"` function until it exhaust all the nodes. Then, the Service Manager loops on the service template looking for a node of type `"OutputConfig"` and parses the JSON contained within the node template key `"func"`.

Finally the Service Manager loops on the service template looking for a node of type `"ErrorCatch"` and when found it encapsulates the `"func"` function in a Node.JS error catch prototype of type `".catch( function(error) {…} )"` and it appends it to the variable which represents the `"main.js"` script.

After the parsing of the service template, the Service Manager writes the `"index.js"` scripts for both the functions and services to the filesystems in the relative directories and finally configures the inputs and outputs of the flow.

Concerning the flow outputs, in the `"filter"` JSON object of the output configuration has a mandatory `filterkey` with an associated `filtervalue`. This is an `"event.action.type"` key which contains the type of action to be send back to the service triggering the flow, e.g. an FCScanner Service.

The Service manager therefore looks in the Service Repository via the Service Access Interface if it exists a service which has a particular input (*hasInput*) name (*hasName*) matching the provided `filtervalue`. If this service is provided (*isOfferedBy*) by a VE, the Service Manager also checks if there exists a VE for the associated (*hasOwner*) user provided back in the Service Request Front-End form.

Concerning the flow inputs, the Service Manager parses keys and values of the `"filter"` JSON object in the input configuration and adds an encapsulating `if((filterkey1 === 'filtervalue1')&&(filterkey2 === 'filtervalue2') …) {…}` to the `"main.js"` script. This has the objective of adding the possibility to prevent the execution of the script in case of a triggering event which does not contain certain particular custom values which are sent by the service triggering the flow. In addition, similarly to the case of the output configuration, the Service manager needs to have at least a mandatory `"event.action.type"` key which is used to look in the Service Repository via the Service Access Interface if it exists a service which has a particular output (*hasOutput*) name (*hasName*) matching the provided `filtervalue`. Then, if this service is provided (*isOfferedBy*) by a VE, the Service Manager also checks if there exists a VE for the associated (*hasOwner*) user provided back in the Service Request Front-End form.

Additionally, the Service Manager parses keys and values of the "context" JSON object in the input configuration and adds an initialization at the beginning of the variable representing the "main.js" for each of those parameters in the form of:

- `context.contextkey1 = 'contextvalue1';`
- …

The Service Manager then adds two ".then()" sections to the variable for the "main.js". The first one it sends the "event.action.type" of the output configuration by triggering an event which will be sent to the relevant service. Both in the case of either an error (caught by the ".catch()" section) or in the case of a correct execution of the flow the action type is sent to the relevant service, but it will carry different payloads, which can be fully customized. The second and final one is in charge of ending the execution of the flow by triggering a "done()" call, which is specific for the Workflow Enforcer and which terminates the execution of the flow and release the occupied resources.

Lastly, the Service Manager adds some "require()" variables for the inclusion of the relative services and functions scripts in the two subdirectories, in the form of:

- `var tagitsmartService = require('./resolved-services');`
- `var tagitsmartFunction = require('./included-functions');`

Finally, it encapsulates the code in the "main.js" in a function which is specific for the Workflow Enforcer which is called whenever an action event is triggered, namely "function onActionCreated(event){…}".

The final "main.js" is then written to the filesystem and from the input configuration (see Section 4.1.1) the "dependencies" section is also written in the filesystem in a customary "package.json" file, which is needed for any javascript application to specify dependencies.

Then the Service Manager prepares a "workflow.zip" archive including all the generated files which are needed by the Workflow Enforcer.

An application placeholder with the "name" of the flow (provided in the input configuration) is therefore created and an HTTP PUT call to the Workflow Enforcer with the zip file as a form parameter is therefore executed in order to inject the bundle script.

### *4.2.2 Workflow enforcer*

The Workflow Enforcer is a component defined in the TagItSmart platform for the execution of the workflow defined in the Service Templates. In the previous section, the procedure for the creation of the workflows was presented. The outcome of this process is a script associated to the execution of the Service Template.

The Workflow Enforcer is then responsible for the execution of this script. It will act as the running environment for the workflow that was defined by the developer and resolved by the Service Manager. Such component needs to enable the deployment and running environment on demand of the code generated from the template. As it was explained before, we chose javascript as the language of the template, as it is easy for Web developers to understand, due to its wide adoption. It is also directly reusable in the implementations of the Workflow Enforcer

based on two of the available IoT platforms in the project, Microsoft Azure and EVRYTHNG Platform, which are explained in more detailed below.

The workflow enforcer will therefore store the resolved template, as a javascript programme, that can be run on demand in the underlying IoT Platform at scale when a request is issued by the Service Manager. The Workflow Enforcer will provide the Runtime Environment that will deploy the correspondent script when the Service Template is invoked. These scripts are written in Node.js and may use NPM (node package manager) packages via a conventional package.json file.

**EVRYTHNG Workflow Enforcer (WFE)**

The EVRYTHNG Workflow Enforcer is built on top of one of the functionalities provided by the EVRYTHNG Platform, which enables the execution of custom code as the result of an event/action/service being invoked in the platform.

Generically, it allows javascript programmes to be attached to endpoints, which serves as the link with the Service Templates resolved by the Service Manager. The underlying infrastructure, based on Amazon Lambda (https://aws.amazon.com/lambda/), provides the Runtime Environment needed to execute the scripts at scale.

Once configured, the scripts will run on demand whenever the service they are attached to is invoked. As the script is written in a full fledged language like Javascript, this allows complex logic to occur in the context of virtually any workflow, driving rich integrations.

One of the benefits of using Amazon Lambda is that it takes care of the provisioning and management of the servers needed to run the code, while the Workflow Enforcer focuses on providing the logic and the acces interface (Web API) to store and launch these scripts representing the Service Templates.

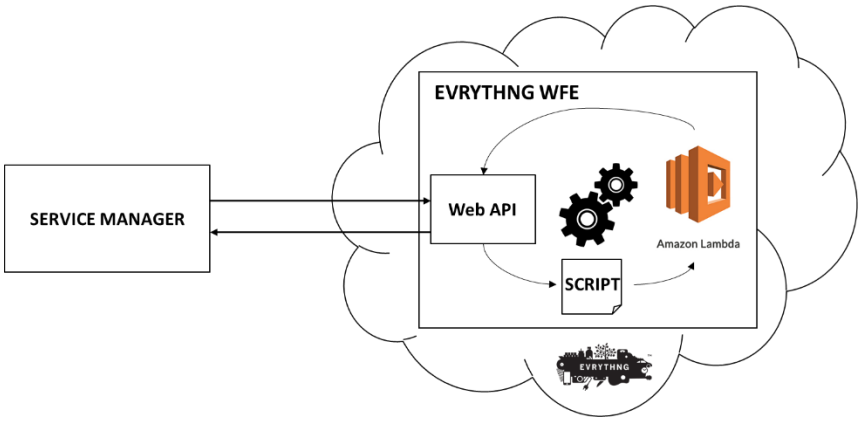


**Figure 25. EVRYTHNG Workflow Enforcer**

## Azure Workflow Enforcer (WFE)

Azure Workflow Enforcer (WFE) is created based on Microsoft Azure Functions component, which enables execution of the Javascript code created by the Service Manager. WFE is composed of two components: Webservice and Azure Functions. Webservice is developed using ASP.NET and this Web API is used as an entry point for the WFE. The role of this component is to take the workload and adapt it in the format readable by the Azure Functions. Deployment of new Azure Functions is done based on Javascript code received from the Service Manager. After successful deployment, Web API has RESTful endpoint deployed which respresents newly created Azure Function.

The result of the call to this endpoint will return Azure Function result, which in this scenario represents the result of the Azure WFE required by the rest of the platform.

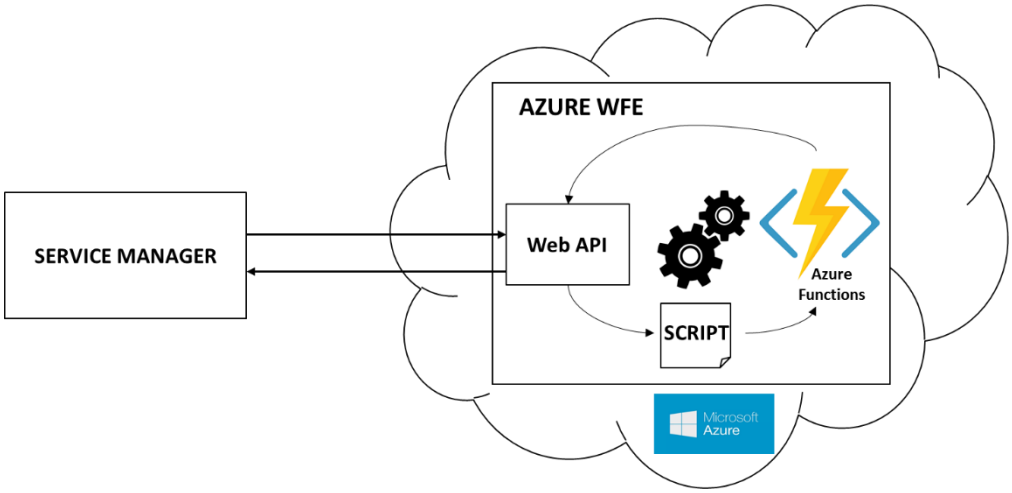


**Figure 26. Azure Workflow Service Enforcer**

## Section 5 - Service Composition in Action

In this section, we report a complete example for the Composition as a Service framework developed for TagItSmart that corresponds to the instantiation of the services required for scanning a Smart Tag associated to an Ice Cream physical entity and visualizing product associated information depending on its status.

For this example we consider two users of the platform:

- ➔ A platform user called "*Stelios*" which has administrative and developer privileges and is in charge of guiding the service composition from scratch, thus discovering, preparing and orchestrating the required services.
- ➔ A user "*Riccardo*" which scans the SmartTag found on an Ice Cream to receive ad-hoc information about the product scanned, therefore using the composed services.

Before actually starting the composition process, Stelios starts by logging onto the platform in the Templates Front End in order to prepare a template for an "*Ice Cream Scan*" flow. Stelios therefore provides the credentials in the form of Figure 3 and its credentials are checked against the ones stored in the User Repository (see Figure 19) with a query from the Templates Front-End via the User Authentication/Access Control module to the User Access interface. In case the user with an email s.georgoulas@surrey.ac.uk is present in the User Repository, the password "*testpwd*" is matched against its secured hashed stored version.

Once logged into the platform, Stelios prepares and connects into a flow all the necessary nodes by either picking from the ones already available and developed by other developers of the platform or also by customizing them in their properties according to the needs of the flow to be made available.

In the Ice Cream Scan example, the flow of Figure 4 involves:

- an InputConfig Node to prepare the service inputs for the flow, i.e. the involved FC Scanner.
- a ContextChecker custom function which checks the integrity and correctness of the runtime context passed by the FC Scanner.
- an Authentication service which checks the credentials passed by the FC Scanner piggybacking on the service inputs.
- an Encoding service which decodes the SmartTag and validates the correctness of the scanned product against the configured type, i.e. an Ice Cream.
- a VELifecycleManager service which is in charge of updating the VE composed by the Smart Tag and Physical Entity.
- a StreamProcessing service, which is in charge of transforming the decoded tag arguments, i.e. "*cold_chain_broken = true/false*" and "*consumption_temperature_ok = true/false*" into messages for a DisplayVE.
- a Notification service which listens on a DisplayVE visualizing the required product information coded into the messages coming from the StreamProcessing service.
- a Feedback action custom function which prepares the feedback message for the service output, in case of a correct execution of the flow.
- an ErrorCatch node which prepares the feedback message for the service output, in case of an error within the execution of the flow.

- an OutputConfig Node to prepare the service outputs for the flow, i.e. the FC Scanner required feedback.

After having created this service template for the flow, Stelios stores it into the Service Template Repository via th Service Access Interface as shown in Figure 5. This is achieved by selecting from the menu entries the "*Export to TagItSmart*" tab.

The same user Stelios, now in the role of a user of the platform for Composition as a Service, accesses the Service Request Front-End with his credentials in the same way, following the same procedure mentioned for the Templates Front End (see Figure 6).

Stelios is then presented with the interface of Figure 7, where the available service templates in the platform are listed. In this case there's only one service template corresponding to the Ice Cream Scan flow of above. Stelios therefore inputs the Service Name "*Ice Cream Scan*", the username (i.e. the email r.pozza@surrey.ac.uk) for the user "*Riccardo*" as the user of the FC-Scanner and then leaves empty the location, so that no geographical location filter is used.

Once submitted the request to the Service Manager, the composition process starts with the retrieval of the "*Ice Cream Scan*" template reported in Section 4.1.1 via the Service Access Interface and the Service Template Repository with a match by name.

After having found a service template, the Service Manager looks for the availability of a user Riccardo in the User Repository in the same way the Templates Front End looked for a user Stelios. In particular, the procedure followed both for the user Stelios in the aforementioned authentication steps and for the user Riccardo inputed in the Service Request Front End, is the following: first the User Authentication/Access Control module is delegated the credentials in the form of email and eventual password, then the User Repository is accessed via the User Access Interface in order to look for an entry which has a specific email field (*hasCharacteristic*) matching either r.pozza@surrey.ac.uk or s.georgoulas@surrey.ac.uk. In case a password is provided a check for a password (*hasCharacteristic*) matching an hashed password is carried out (see Section 4.1.4).

The Service Manager then resolves the service templates parameters for each of the services mentioned above. In particular, for the Authentication, the Encoding, the VELifecycleManager and the StreamProcessing services the Service Manager looks via the Service Access Interface into the Service Repository for a service which includes a function (*includesFunction*) which has a name (*hasName*), matching, respectively, "*Authentication*", "*Encoding*", "*VELifecycleManager*" and "*StreamProcessing*". The Service Manager then looks for the parameters provided into a "*options*" JSON object and substitute the parameters with the "*to-be-resolved*" placeholders.

Specifically, for the Authentication, the Encoding, the VELifecycleManager and the StreamProcessing services it replaces the key-value pair "*uri: 'url-to-be-resolved'*" with the key-value pairs ( *hasLogicalLocation and hasURI* ) respectively of "*uri: 'https://api.evrythng.com/auth/evrythng'*", "*uri: 'https://api.evrythng.com/scan/identifications'*", "*uri: 'https://api.evrythng.com/products'*" and "*uri: 'https://131.227.92.228:4321/process'*" (see Section 4.1.2).

In some cases, such as for the Notification service in the Ice Cream example, the Service Manager finds that the service is offered by (*isOfferedBy*) a Virtual Entity, i.e. a DisplayVE. The

Service Manager then queries through the VE Access Interface the VE Repository for the presence of a DisplayVE (see Section 4.1.3) which offers a service (*offersService*) Notification Service (see Figure 17) and also matches the user (*hasOwner*) Riccardo, specified in the Service Request Front End. When found the correct VE, the Service Manager gets the related "Nofitication Service" (*offersService*) and which has an endpoint (*hasLogicalLocation* and *hasURI*) "*https://mwcmicro.screenlyapp.com*" and substitutes the endpoint in a similar way as for the other services.

Then, based on the flow Input and Output configuration, the Service Manager looks in the Service Repository via the Service Access Interface if there exists a service which has a particular input (*hasInput*) or output (*hasOutput*) name (*hasName*) matching, respectively *"smartTagReadDone"* and *"smartTagRead"*. The Service Manager, in the Ice Cream case therefore finds a *"FCScanService"* (see Figure 14) and also finds that the service is offered by (*isOfferedBy*) a "*FCScannerVE*". Similarly to the case of the Notification Service, the Service Manager then queries through the VE Access Interface the VE Repository for the presence of a FCScannerVE which matches the user (*hasOwner*) Riccardo, specified in the Service Request Front End.

Finally, based on the actual resolved services and service template the Service Manager prepares a *"workflow.zip"* archive including all the necessary files required by the Workflow Enforcer for the execution of the service composition. This zip file containing the Ice Cream Scan flow is therefore sent to the Workflow Enforcer which creates an execution engine for the source files and listens for events corresponding to scanning actions sent by the user Riccardo.

When the user Riccardo scans a QR-code representing an Ice Cream Product, the Workflow Enforcer acts as the "middle-man" and orchestrates the requests to the actual resolved services. In particular, following a "*smartTagRead*" action, the email "r.pozza@surrey.ac.uk" and password "*test123*" of user Riccardo are sent along with base64 encoded image data representing the Smart Tag to the Workflow Enforcer. Based on the InputConfig filters, only if the action is a "*smartTagRead*" the flow will be executed. The credentials are then stored in an internal "*context*" JSON object along with the image data and with a parameter from InputConfig representing the Ice Cream product ID to validate against once decode.

At the beginning, the authentication service takes the input credentials and if successful, returns an api key to be used for authorization to the subsequent services. Then the encoding service decodes the image representing the Smart Tag and outputs a "*cold_chain_broken = true/false*" and "*consumption_temperature_ok = true/false*" representing, respectively if the ice cream ever reached in his lifetime a temperature higher than the one set as a the cold_chain threshold and if the ice cream has a consumption temperature which is ok at the moment. In addition the decoding service returns a product ID which is validated against the one of an Ice Cream provided in the aforementioned InputConfig parameter.

Then a VELifecycleManager service is called by the flow to update the VE corresponding to the SmartTag and Physical Entity of Figure 18 and Figure 21 with the values obtained after the successful decoding of the Smart Tag.

Moreover, a Stream Processing service is called to process the same two values and obtain a output message for the Notification Service which depends if either one or both of the arguments are true or false. For example if the consumption_temperature_ok=true and cold_chain_broken=false it means that the message to display is that the Ice Cream is in

perfect conditions and has never reached higher temperatures than the ones measurable by the Smart Tag.

Finally the Notification Service visualizes this message on a DisplayVE (i.e. a raspberry pi with an attached display) and the flow returns a "*smartTagReadDone*" action to the FCScanner, to ensure the correctness of the execution of the flow. This action contains the "*workflowOutput*" parameters either for the case of a correct execution of the flow or for the case of any error in the execution of the flow (e.g. failed authentication, wrong decoding, etc…).

Below are reported examples of all the files required by the WorkFlow Enforcer for enforcing the composition request in this case of an "*Ice Cream Scan*" flow.

**main.js**

```
var tagitsmartService = require('./resolved-services');
var tagitsmartFunction = require('./included-functions');

function onActionCreated(event) {

  if( (event.action.type === '_smartTagRead')) {
    var context = event.action.customFields;
    context.allowedProductID = 'U3gECAt8BDsRQKRRwDAA8t2n';
    logger.info('Starting the IceCream-Scan workflow with ' + JSON.stringify(context, null, 4));

    tagitsmartFunction.ContextChecker.check(context)
    .then(function () {
      // Authentication Service
      logger.info('Authentication!');
      return tagitsmartService.Authentication.validate(context)
        .then(function (result) {
          logger.info('Authentication Successful!');
          context.userApiKey = result.evrythngApiKey;
        }, function (error) {
          logger.info('Authentication Error!');
          throw error;
        });
      // Authentication Service
    })
    .then(function () {
      // Encoding Service
      return tagitsmartService.Encoding.decode(context)
        .then(function (results) {
          //Decoding successful
          logger.info('Decoding Successful!');
          context.product = results[0].results[0].product;
          logger.info(JSON.stringify(context.product, null, 4));
          logger.info(JSON.stringify(results, null, 4));

          if (!context.updates) {
```

```
            context.updates = [];
        }

        //update values
        for (let result of results) {
            if (result.meta.method === 'ir') {
                context.updates.push({
                    key: 'consumption_temperature_ok',
                    value: true
                });
            }
            else if (result.meta.method === '2d') {
                context.qrValue = result.meta.value.slice(-2);

                return app.product().read({
                    params: {
                        filter: 'tags=' + context.qrValue
                    }
                }).then(function (product) {
                    context.product = product[0];
                    logger.info('Product Found by Identifier');

                    // Check product is ice cream
                    if (context.product.id !== context.allowedProductID){
                        throw new Error('Unrecognized Type of Product');
                    }
                    // Checked product is ice cream
                    context.updates.push({
                        key: 'cold_chain_broken',
                        value: !(context.product.customFields[context.qrValue.toString()] ===
'cold_chain_ok')
                    });
                });
            }
        }
    }, function (error) {
        //Decoding error
        logger.info('Decoding Error!');
        throw error;
    });
  // Encoding Service
})
.then(function () {
    //VELifecycleManager Service
    return tagitsmartService.VELifecycleManager.update(context)
        .then(function (result) {
            //Update successful
```

```
            logger.info('VE Update Successful!');
            logger.info(JSON.stringify(result, null, 4));

            context[result[0].key] = result[0].value;
            context[result[1].key] = result[1].value;

        }, function (error) {
            logger.info('VE Update Error!');
            throw error;
        });
    //VELifecycleManager Service
})
.then(function () {
    //Stream Processing Service
    return tagitsmartService.StreamProcessing.process(context)
        .then(function (result) {
            logger.info('Stream Processing successful with output!' + JSON.stringify(result.output1));
            context.message = result.output1;
        }, function (error) {
            logger.info('Stream Processing Error!');
            throw error;
        });
    //Stream Processing Service
})
.then(function () {
    //Notification Service
    return tagitsmartService.Notification.notify(context)
        .then(function (result) {
            //Notification successful
            logger.info('Notification to Screenly Sent Successfully!');
        }, function (error) {
            logger.info('Notification to Screenly Error!');
            throw error;
        });
    //Notification Service
})
.then(function () {
    return tagitsmartFunction.FeedbackAction.configure(context)
})
.catch(function (error) {
     logger.error("Error executing the flow " + JSON.stringify(error, null, 4));

    // Defined in Node-RED
    context.workflowOutput = {
        product: context.product.id,
        customFields: {
            error: error
```

```
            }
        };
        // Defined in Node-RED
    })
    .then(function () {
        return app.action('_smartTagReadDone').create(context.workflowOutput)
            .then(function (action) {
                logger.info("Workflow output sent!");
            }, function (error) {
                logger.info(JSON.stringify(error));
            });
    })
    .then(function(){
        logger.info("Ending the workflows!");
        done();
    });

  }

}

exports.onActionCreated = onActionCreated;
```

**package.json**

```
{
  "dependencies": {
    "evrythng-extended": "^4.1.0",
    "request": "*",
    "request-promise": "*"
  }
}
```

**resolved-services/index.js**

```
var Authentication = require('./Authentication.js');
var Encoding = require('./Encoding.js');
var VELifecycleManager = require('./VELifecycleManager.js');
var StreamProcessing = require('./StreamProcessing.js');
var Notification = require('./Notification.js');

exports.Authentication = Authentication;
exports.Encoding = Encoding;
exports.VELifecycleManager = VELifecycleManager;
exports.StreamProcessing = StreamProcessing;
exports.Notification = Notification;
```

**resolved-services/Authentication.js**

```
exports.validate = function(context){
  var request = require('request-promise');
  var options = {
    method: 'POST',
    uri: 'https://api.evrythng.com/auth/evrythng',
    body: context.credentials,
    json: true, // Automatically stringifies the body to JSON
    headers: {
      'Content-Type': 'application/json',
      'Authorization': context.appApiKey
    }
  };
  return request(options);
}
```

**resolved-services/Encoding.js**

```
exports.decode = function(context){
  var request = require('request-promise');
  var options = {
    method: 'POST',
    uri: 'https://api.evrythng.com/scan/identifications' + '?filter=method=2d,ir&type=qr_code,image',
    body: { image: context.image },
    json: true, // Automatically stringifies the body to JSON
    headers: {
      'Content-Type': 'application/json',
      'Authorization': context.appApiKey
    }
  };
  return request(options);
}
```

**resolved-services/VELifecycleManager.js**

```
exports.update = function(context){
  var request = require('request-promise');
  var options = {
    method: 'PUT',
    uri: 'https://api.evrythng.com/products' + context.product.id + '/properties',
    body: context.updates,
    json: true, // Automatically stringifies the body to JSON
    headers: {
      'Content-Type': 'application/json',
      'Authorization': context.userApiKey
    }
  };
  return request(options);
```

```
}
```

**resolved-services/StreamProcessing.js**

```
exports.process = function(context){
   var request = require('request-promise');
   var options = {
      url: 'https://131.227.92.228:4321/process',
      method: 'POST',
      json: true,
      rejectUnauthorized: false,
      requestCert: true,
      agent: false,
      body: {
         arg1: context.cold_chain_broken,
         arg2: context.consumption_temperature_ok
      }
   };
   return request(options);
}
```

**resolved-services/Notification.js**

```
exports.notify = function(context){
   var request = require('request-promise');
   var options = {
      method: 'POST',
      uri: 'https://mwcmicro.screenlyapp.com',
      form: {sku: context.message},
   };
   return request(options);
}
```

**included-functions/index.js**

```
var ContextChecker = require('./ContextChecker.js');
var FeedbackAction = require('./FeedbackAction.js');

exports.ContextChecker = ContextChecker;
exports.FeedbackAction = FeedbackAction;
```

**included-functions/ContextChecker.js**

```
exports.check = function(context){
   //This function could define any transformations or checks needed in the context
   // e.g. check that all the inputs are available, combine EVT info in the context
   return new Promise(function (resolve, reject) {
      if (context !== null) {
         resolve(context);
      } else {
         reject("Context not valid");
```

```
    }
  })
}
```

**included-functions/FeedbackAction.js**

```
exports.configure = function(context){
  //This function could define any transformations or checks needed in the context
  // e.g. generate the output of the flow
  context.workflowOutput = {
    product: context.product.id,
    customFields: {
      product: context.product
    }
  };

  logger.info(JSON.stringify(context.workflowOutput));

  return new Promise(function (resolve, reject) {
    if (context.workflowOutput !== null) {
      resolve(context);
    } else {
      reject("Workflow output not valid");
    }
  });
}
```

## Section 6 - Conclusions

In this deliverable we presented the TagItSmart approach towards service composition. The core components (Service Manager, Workflow Enforcer) together with needed repositories and interfaces (the latter to be described in detail in D1.3 [15] due in M24) allow for exposing the TagItSmart platform capabilities for use in service requests, their combination guided by suitable service templates that developers can insert through a user friendly interface and the issuing and execution of service requests even from users not aware of low level technology details.

E.g. a user issuing the service request of Section 5 does not need to know that the request needs to involve a specific Stream Processing service or where this service is listening to; the decomposition and resolution of services is something transparent to them, as is also the runtime execution. Together with measures and enablers developed in WP2/3, all these also take place in a secure and privacy preserving manner, increasing the confidence of users in the platform.

Next steps in this task include enhancement of the Service Manager with advanced service composition making based on multiple suitability criteria, assisted by properties in the semantic representation of VEs, users and services and finalization of an adaptation layer for seamless integration also with the Microsoft Azure workflow enforcer allowing for the service composition part of the TagItSmart platform to be usable in two major production grade environments, in order to facilitate and increase the real-life exploitation opportunities.

## Section 7 - References


1. IoT-A D1.5 deliverable “Final architecture reference model for the IoT v3”, July 2013
2. EU iCore project, http://www.iot-icore.eu/
3. EU-JP iKaaS project, www.ikaas.com
4. EU IOT.est project, http://ics-iot.weebly.com/iotest.html
5. EU COSMOS project, http://iot-cosmos.eu
6. EU COMPOSE project, http://www.compose-project.eu/
7. Node-RED website, https://nodered.org/
8. W3C website, https://w3.org
9. TagItSmart D1.2 deliverable, “Initial TagItSmart Reference Architecture and API Description”, January 2017.
10. MongoDB website, https://www.mongodb.com/what-is-mongodb
11. W3C website - RDF, https://www.w3.org/RDF/
12. TagItSmart D4.1 deliverable, “Resource and Service modelling Specification”, January 2017.
13. Virtuoso website, https://virtuoso.openlinksw.com/rdf/
14. W3C website - SPARQL, https://www.w3.org/TR/rdf-sparql-query/
15. TagItSmart D1.4 deliverable, “Final TagItSmart Reference Architecture and API description”, due in December 2017.